DSR Module Iteration One Detailed Design

Greg Campbell

May 30, 2003

Contents

1 Introduction

The aim of this document is to detail how the *DSRModule* class will be implemented and how it will fit into the existing PIX framework. Readers should be familiar with the 'DSR Implementation Preliminary Design' ([3]) document.

The main goals of the design are as follows:

- Allow the same code base to be used to provide DSR at different protocol layers (specifically AX.25 and IP).
- Require minimal modifications to allow an existing PIX protocol to use DSR.

Since this is the first iteration of the DSR Module, many DSR features will not be supported. The following list highlights the features that will, and will not be supported in iteration one:

- Route discovery will be implemented.
- Route maintenance will not be implemented. This means that frames sent over a link are assumed to arrive intact at the destination address. Thus, no DSR ACKs or route errors are required. Also, neither the retransmission buffer nor packet salvaging are implemented. This is one of the first things that should change in iteration two.
- All links are assumed to be bidirectional.
- Route requests will be rebroadcast with an exponential backoff algorithm as outlined in [1].
- The Route Cache will be a simple path cache (one entry for each complete path). Later iterations should implement a more powerful link cache.
- This iteration will not cache any overheard route information.
- This iteration will not reply to route requests using cached paths.
- This iteration will not limit the maximum length of paths in route request packets (unless the maximum length of DSR options is exceeded).
- This iteration will not perform automatic route shortening (gratuitous route replies).
- This iteration will not include any external network support.

2 DSR Option Headers

The main DSR draft, [1] outlines how IPv4 extenstion headers should be used to support DSR. Since support for extension headers is not found in link layer protocols (namely AX.25) some modifications must be made if we are to support DSR at the link layer. [3] explains in detail how DSR headers could be handled at the link layer using AX.25. Since one of our main goals is that the *same* code base be used for DSR in both IP and AX.25, the *same* header format for DSR must be used. Thus, IPv4 extensions are not used to support DSR, rather the format presented in [3].

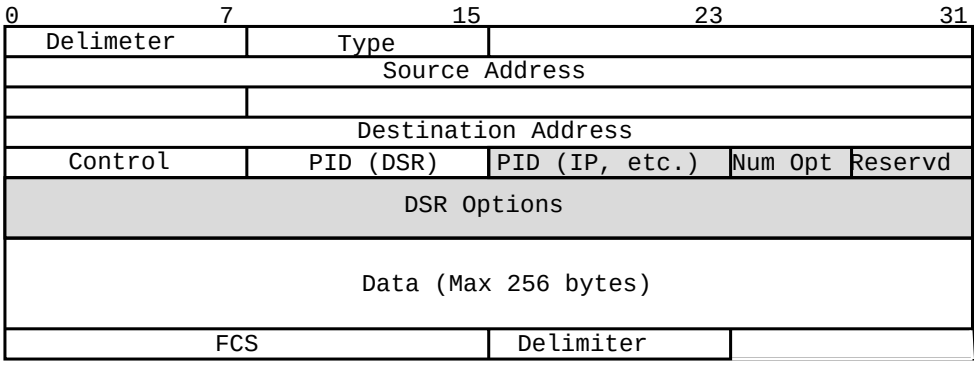

Figure 1: An AX.25 frame containing DSR options.

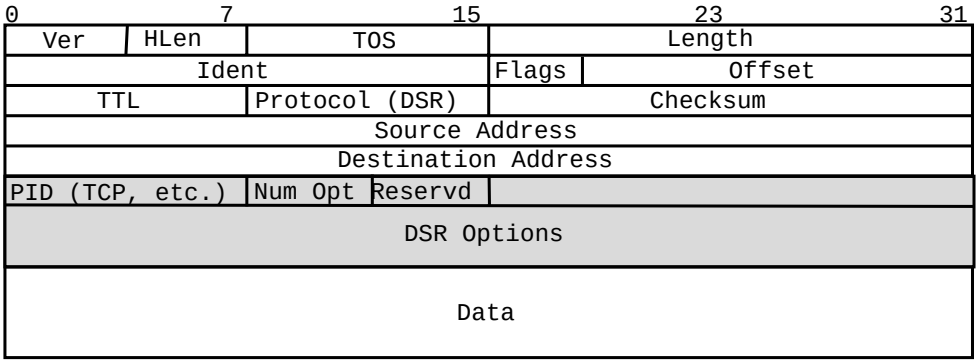

Figure 2: An IPv4 frame containing DSR options.

[3] specifies that, for any AX.25 frame carrying DSR options, the frame format shown in 1 be used. Here, the *PID* indicates the frame has DSR options and the payload consists of the DSR preamble followed by the DSR options and finally the actual payload. All DSR options are defined as the draft ([1]) but with 7 byte addresses.

To use the same code base we just apply this exact format to IP (so IPv4 extensions are no longer used). The only changes that must be made is that all addresses are now 4 rather than 7 bytes, and it is the *Protocol* field that indicates the packet to have DSR options). This is pictured in figure 2.

3 DSR Module Overview

This section provides an overview of the DSR module and how it fits into the PIX framework and existing procotols. This amounts to detailing the public interface that the DSR module provides to protocols and exactly how it is intended to be used.

3.1 DSR Module Setup

The following code shows an example of how to add DSR support to the AX.25 layer (error handling omitted for clarity). Note that some additional code must also be present but the

discussion of this is deferred for section 7.

```
1  AX25Protocol *  ax25 = new AX25Protocol(device, result);
2  AX25DSRModule * ax25DSRModule = new AX25DSRModule(homeAddress, ax25);
3  ax25->xControl(SET_DSR_MODULE, ax25DSRModule, 0);
```

To add DSR support at the IP layer the same format is used except an $IPDSRModule$ would be instantiated and attached to an $IPProtocol$. Note that this requires the protocol to handle the SET_DSR_MODULE message in its $xControl$. For all protocols, this code would do the following:

```
1  int AX25Protocol::xControl(int op, void *buf, uint len)
2     {
3     ...
4     case SET_DSR_MODULE:
5        dsrModule = (AX25DSRModule *)buf;
6        return 0;
7     ...
8     }
```

3.2 Message Reception

The technique for message reception in a protocol supporting DSR is to have code similar to the following in $xDemux$ (error handling omitted for clarity):

```
1  Outcome xDemux(Session *lls, Message *msg)
2     {
3     uchar * hdr = extractMessageHeader(msg);
4     int     actualHLP;
5     if(hdr[PROTOCOL_TYPE] == DSR)
6        {
7        int retVal = dsrModule->processDSROptions(hdr, msg, &actualHLP);
8        if(retVal == DO_NOT_HANDLE) return OK;
9        }
10    ... Do whatever the protocol would normally do to handle a msg ...
11    }
```

Basically, the start of the $xDemux$ remains the same as without DSR - we extract the header. Then, lines 5 to 9 contain the code that must be added to support DSR; if the message is a DSR message, remove and process the DSR options. Note that upon return all DSR options have been removed from msg and $actualHLP$ will be populated with the actual PID of the message (i.e. if this code was at the AX.25 layer, $actualHLP$ would generally indicate IP). There are two outcomes of processing DSR options: no more processing for the packet is required (line 8), or processing of msg should occur as normal for the protocol.

3.3 Message Sending

The technique for message reception in a protocol supporting DSR is to have code similar to the following in a sessions $xPush$ (error handling omitted for clarity):

```
1  Outcome IPSession::xPush(Message *msg)
2     {
3     if(dsrModule != NULL)
4        dsrModule->xPush(sessionHdr, msg);
5     else
6        ... Do whatever normally done to send message ...
7     }
```

Thus, if there is a *DSRModule*, calling that module's *xPush* will do everything required to send a message. If there is no *DSRModule* the message should be handled as normal.

4 *DSRModule* Details

This section gives the detailed design of both the *DSRModule* class and its subclasses. The methods provided by *DSRModule* and the methods overridden by the subclasses are discussed in detail.

4.1 *DSRModule* Heirarchy

Figure 3 shows the general heirarchy that is used to represent a DSR Module. The *DSRModule* base class contains all common code and the subclasses (in this example *AX*25*DSRModule* and *IPDSRModule*) implement the required abstract functions to form concrete *DSRModule* classes. All program flow is controlled by the *DSRModule*. The *DSRModule* calls virtual methods implemented by the subclass as required to perform protocol specific operations. The goal of this model is to keep all functionality in the base class and the subclasses very simple and easy to implement. This allows us to add DSR functionality to new protocols by creating a very simple *DSRModule* subclass with the appropriate methods.

Note that 3 has been highly simplified here (private methods and attributes not shown). The figure is only intended to show the relationship between the *DSRModule* superclass and its subclass and the public interface.

4.2 *DSRModule* Design

Figure 4 shows the complete *DSRModule* class definition. The goal of this section is to document each of the base *DSRModule*'s attributes and methods. The section is divided into three parts: attributes, concrete methods, and abstract methods.

4.2.1 Attributes

homeAddress : The local address of the *DSRModule*. All DSR option addresses use this address to refer to the local protocol stack.

broadcastAddress : The address to be used by the *MSRModule* to broadcast packets. For AX.25 this would be populated to be "CQ" while for IPv4 it would be 255.255.255.255.

protocol : The protocol that contains the *DSRModule*. This is required so the *DSRModule* can refer back to its parent protocol.

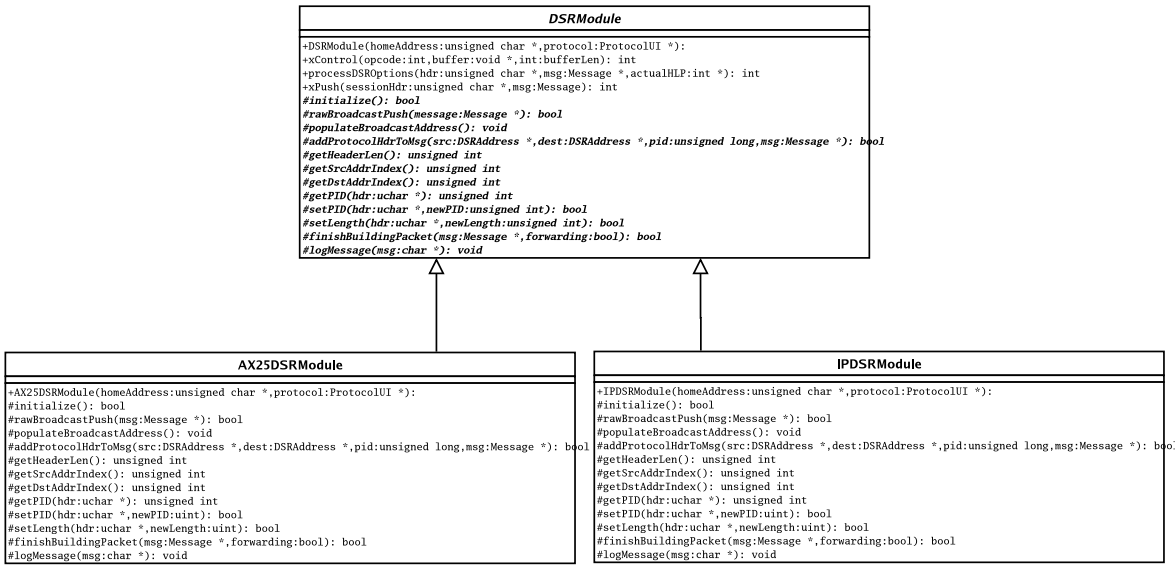

Figure 3: The DSRModule heirarchy.

prototypeDSRPreamble : A prototype DSR preamble. This is the same for all protocols ($DSRModule$ subclasses).

flags : A bitmask of flags for the $DSRModule$. These flags are manipulated by calling $xControl$. Valid flags are as follows:

- $UNIDIRECTIONAL_LINKS$: Set to indicate the DSRModule is operating in a network containing unidirectional links. If not set, the $DSRModule$ will assume that all links are bidirectional.
- $CACHE\_OVERHEARD\_ROUTE\_INFO$: Set to indicate overheard routing information should be used to maintain the route cache. Off indicates overheard routing information should be ignored.
- $ROUTE\_ERRORS$: Set to indicate that route errors should be used when broken links detected. Off indicates route errors are not used. Must be off in the current iteration.
- $REPLY\_TO\_RREQ\_USING\_CACHE$: Set to indicate route requests should be replied to using cached routes if possible. Off indicates no route requests replies from route cache information. Must be off in the current iteration.
- $PREVENT\_RREP\_STORMS$: Set to indicate that route reply storms should be avoid (by delaying sending route replies based on cache information for a length of time proportional to the length of the path to return). Off indicates no route reply storm avoidance. Requires $REPLY\_TO\_RREQ\_USING\_CACHE$ be set. Must be off in the current iteration.
- $PACKET\_SALVAGING$: Set to indicate packet salvaging should be used. Off indicates not packet salvaging. Must be off in the current iteration.
- $GRATUITOUS\_ROUTE\_REPLIES$: Set to indicate that automatic route shortening via gratuitous route replies should be used. Off indicates no gratuitous route replies. Must be off in the current iteration.

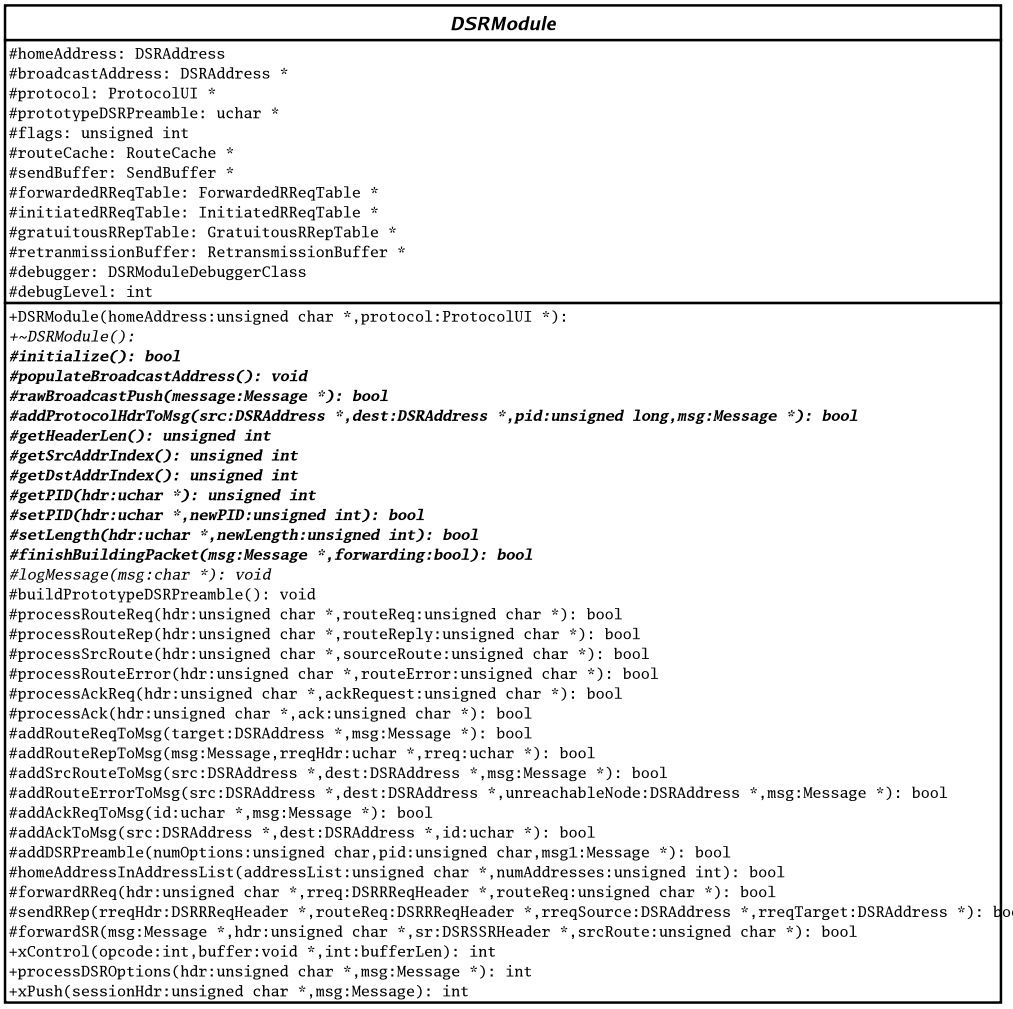

Figure 4: The DSRModule details.

- $INCREASED\_ROUTE\_ERROR\_SPREADING$: Set to indicate route errors should be propogated with new route requests. Off indicates no increased route error propogation. Requires $ROUTE\_ERRORS$ be set. Must be off in the current iteration.
- $QUEUE\_PACKETS\_FOR\_BROKEN\_LINK$: Set to indicate that packets destine for broken links should be queued. Off indicates no queueing of messages destine for a broken link. Must be off in the current iteration.
- $EXTERNAL\_NETWORK\_SUPPORT$: Set to indicate that external networks (as the first or last hop) are supported. Off indicates no support for external networks. Must be off in the current iteration.

$routeCache$: The route cache. This is described in more detail in section 6.4.

$sendBuffer$: The send buffer. This is described in more detail in section 6.5.

$forwardedRReqTable$: The forwarded route request table. This is described in moew detail in section 6.6.

initiatedRReqTable : The initiated route request table. This is described in moew detail in section 6.6.

gratuitousRRepTable : The gratuitous route reply table. Though not used in the current iteration, this is described in more detail in section 6.7.

retransmissionBuffer : The retransmission buffer. Though not used in the current iteration, this is described in more detail in section 6.8.

debugger : The class that is used to debug. The *DSRModule* uses this class to output debug information to the extent determined by *debugLevel*.

debugLevel : The level of debugging to perform. Valid debug levels (in order of increased debug info) are:

- *DEBUG_LEVEL_NONE* : Logs no debug information.
- *DEBUG_LEVEL_LOG_PACKETS* : Logs debug information about the sending and receiving of packets.
- *DEBUG_LEVEL_FULL_TRACE* : Logs complete debug information about each method called in the handling of each packet.

4.2.2 Concrete Methods

buildPrototypeDSRPreamble : Constructs a prototype DSR preamble and stores in *prototypeDSRPreamble*. Anytime a DSR preamble needs to be added to a message, this prototype is copied in and then modified as required.

processRouteReq : Called to process the passed DSR route request.

processRouteRep : Called to process the passed DSR route reply.

processSrcRoute : Called to process the passed DSR source route.

processRouteError : Called to process the passed DSR route error. Should log an error in the current iteration.

processAckReq : Called to process the DSR request for acknowledgement. Should log an error in the current iteration.

processAck : Called to process the passed DSR acknowledgement. Should log an error in the current iteration.

addRouteReqToMsg : Add a route request for the passed *target* to the *msg*

addRouteRepToMsg : Add a route reply to the *rreq* route request to *msg*. *rreqHdr* points to the protocol header of the route request that is triggering the route reply.

addSrcRouteToMsg : Add a source route from *src* to *dest* to the message. Returns *false* if no such path is found in the route cache.

addRouteErrorToMsg : Add an error with the passed information to *msg*. Should log an error in the current iteration.

addAckReqToMsg : Add an acknowledgement request to *msg*. Should log an error in the current iteration.

addAckToMsg : Add an acknowledgement to *msg*. Should log an erro in the current iteration.

addDSRPreamble : Add the DSR preamble with the indicated information to the message.

homeAddressInAddressList : Return whether the passed list of addresses contains the *homeAddress* of the *DSRModule*.

forwardRReq : Called to forward the passed route request i.e. add the *homeAddress* to the route in the route request and then broadcast the route request.

sendRRep : Send a route reply for the passed route request. The home node may be the target of the route request, or it may be responding using cached information; either way this function is resposible for building and sending the route reply back to the originator of the route request.

forwardSR : Forward the passed source route to the next hop destination.

xControl : Used to control the *DSRModule*. Valid *opcodes* are all flags (see 4.2.1) with '*SET*' prepended (i.e. *SET_ROUTE_ERRORS* etc.):

processDSROptions : Called by a protocol to process a packets DSR options. Returns *DSR_IGNORE_PACKET* to indicate the protocol should ignore the packet or *DSR_OK* to indicate the protocol should continue handling it as normal. When this returns, the DSR options have been removed from the message, the actual PID of the message has been copied into *hdr*, and the length field of *hdr* has been updated if appropriate.

xPush : Called by a session to send a packet when DSR is being used. This method performs whatever actions are required (including route discovery) to send the packet. Upon successful return the calling session can assume the packet has been transmitted (or will be once an appropriate route is discovered).

4.2.3 Abstract Methods

initialize : A pure virtual method that subclasses use to perform specific operations. For AX.25, not much is required, but for IP this method would create a lower level broadcast session to use for transmission of packets.

populateBroadcastAddress : Populates *broadcastAddress* with whatever the broadcast address is for the specific protocol.

rawBroadcastPush : A pure virtual method that subclasses call to broadcast a packet unmodified to the the lower level of the protocol stack. For AX.25, this would calls the *AX25Protocol* :: *xControl* method for the *DSR_MODULE_RAW_PUSH* that would call *sendto* on the raw AX.25 socket. For IP, this would call *xPush* on the *IPDSRModule*'s broadcast session.

addProtocolHdrToMsg : Add the proper protocol (AX.25 or IP) header to the message based on the passed information.

getHeaderLen : Returns the length of a header for the protocol the *DSRModule* is operating in. For AX.25 this is 17. For IP, this is 20 bytes (assuming there are no extension headers).

getSrcAddrIndex : Returns the (0 indexed) index of the start of the source address in the header for the protocol the *DSRModule* is operating in. For AX.25 this is 1. For IP this is 11.

getDstAddrIndex : Returns the (0 indexed) index of the start of the destination address in the header for the protocol the *DSRModule* is operating in. For AX.25 this is 8. For IP this is 15.

getPID : Returns the value of the 'PID' or 'Type' field in the passed header for the protocol the *DSRModule* is operating in.

setPID : Set the value of the 'PID' or 'Type' field of *hdr* to *newPID* for the protocol the *DSRModule* is operating in.

setLength : Set the value of the 'Length' field of *hdr* to *newLength* for the protocol the *DSRModule* is operating in. For AX.25 this just returns *true* since there is no 'Length' field in the AX.25 header. For IP, this must actually update the field.

finishBuildingPacket : This method is called right before a message is sent by the *DSRModule*. *forwarding* indicates whether the message is simply being forwarded or is a new message. For IP this would be used to decrement the TTL and calculate the checksum. For AX.25 this would be used to calculate the FCS (if we were using it). For all protocols this would have to set the 'Length' field to be the length of the packet *including* the DSR options (except for AX.25 which doesn't have a length field).

logMessage : Log the passed message.

4.3 *DSRModuleShell*

This section describes a 'kludge' that was used to get around a problem caused by using generative programming to generate the *DSRModule*. The problem is that some classes (like the *SendBuffer*) require a pointer to the *DSRModule* so they can perform various callbacks. But, since the *DSRModule* is generated, other classes would have to know the template parameter used to construct the *DSRModule* in order to keep a pointer to the *DSRModule*. TO avoid this, a non-template class, *DSRModuleShell*, is defined which merely declares all the public *DSRModule* methods as pure virtual methods. *DSRModule* then derives from *DSRModuleShell* and implements the public methods. Helper classes that need to call function on the *DSRModule* can store a pointer to the non-template *DSRModuleShell* and so avoid the need to know the *DSRModule* template parameter to interact with the *DSRModule*.

5 *DSRModule* Subclass Design

This section goes into details on the design of subclasses of *DSRModule* In particular, subclasses for support of DSR at the AX.25 and IP level are discussed: *AX25DSRModule* and *IPDSRModule*. Note how little is required of the subclasses - all the real work is contained in the base *DSRModule*.

5.1 *AX25DSRModule* Design

Section 4.2.3 already describes what all abstract methods should do for a DSR module in the AX.25 protocol. No other attributes or methods should be required by the *AX25DSRModule*. So, figure 5 shows the detailed *AX25DSRModule*.

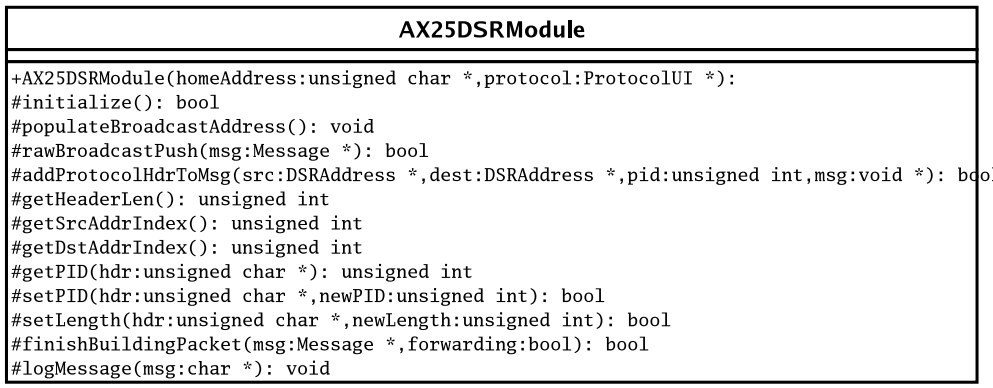

Figure 5: The AX25DSRModule details.

5.2 $IPDSRModule$ Design

Section 4.2.3 already describes what all abstract methods should do for a DSR module in the IP protocol. While $AX25DSRModule :: rawPush$ can directly call $AX25Protocol :: xPush$ to send a packet unmodified down the stack, $AX25DSRModule :: rawPush$ must use a $xPush$ on a broadcast session it has previously opened on the lower level protocol. Thus, $IPDSRModule$ requires the creation, management and destruction of this lower level broadcast session. Figure 6 shows the detailed $IPDSRModule$. The following additions from $AX25DSRModule$ were added:

broadcastSession : The broadcast session that is used by *rawPush* to broadcast DSR packets.

broadcastParticipant : Temporarily holds the participants used to create the broadcast session. This is required because the participants are passed to the constructor but required by *initialize* which accepts no arguments. Therefore, for this information to be available when *initialize* gets called, it must be saved.

$IPDSRModule()$: As mentioned, the constructor is now passed a Participant structure to use in opening the lower level broadcast session.

6 Helper Classes

This section details the various helper classes that are used by the $DSRModule$. Each of these helper classes is an abstract base class that must be made concrete by a subclass. Each section first details the abstract methods defined by the base class and then details an implementation of a subclass (if applicable to the first iteration).

6.1 $DSRErrorHandler$

The $DSRErrorHandler$ class is very simple class from which many of the other helper classes derive. This class provides a single error buffer which is intended to hold the string representation of the last error that occurred in the derived class. Figure 7 shows the details of $DSRErrorHandler$ which are now described in detail:

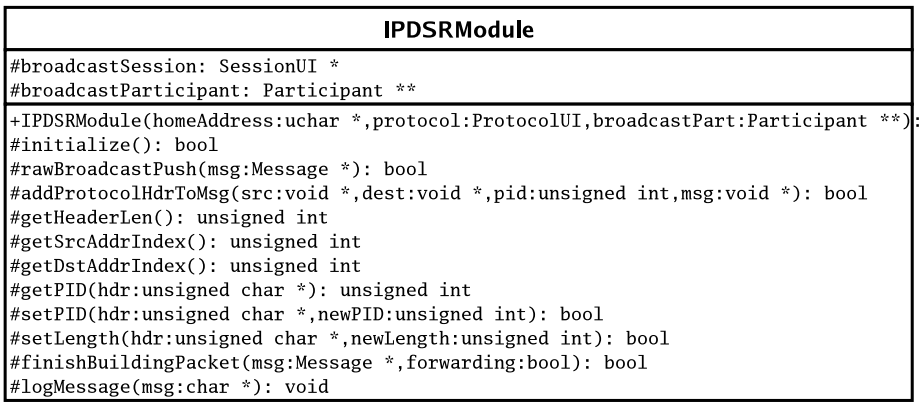

Figure 6: The IPDSRModule details.

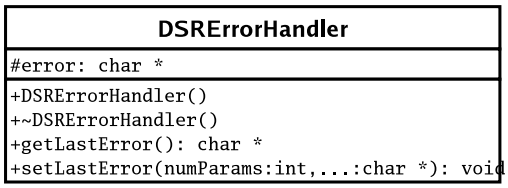

Figure 7: The *DSRErrorHandler* details.

error: Holds the last error that occurred (or NULL if there is no last error).

DSRErrorHandler : The constructor.

$\sim$ *DSRErrorHandler* : The destructor.

getLastError : Returns the current contents of *error*.

setLastError : Sets *error* to a concatenation of all passed strings. Note that this function accepts a variable number of parameters - the caller must specify the number of arguments as the first argument of the call.

6.2 *ObjectPtr*

The sole purpose of the *ObjectPtr* template is to manage a pointer to an object in such a way that it can be used in STL containers. The reason this is required is that if object pointers are used in stl containers they are not stored/copied properly within the container. Anywhere STL containers are used to hold object pointers, the *ObjectPtr* template is used. Figure 8 shows the self explainatory layout of the *ObjectPtr* class.

6.3 *DSRAddress*

The *DSRAddress* class has already come up many times in the description of the *DSRModule* methods. *DSRAddress* is a base class that is used to represent addresses of different length. *DSRAddress* can be used to manipulate any type of address with one restriction: the size of the

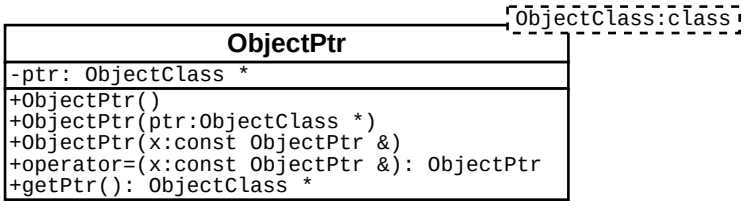

Figure 8: The *ObjectPtr* details.

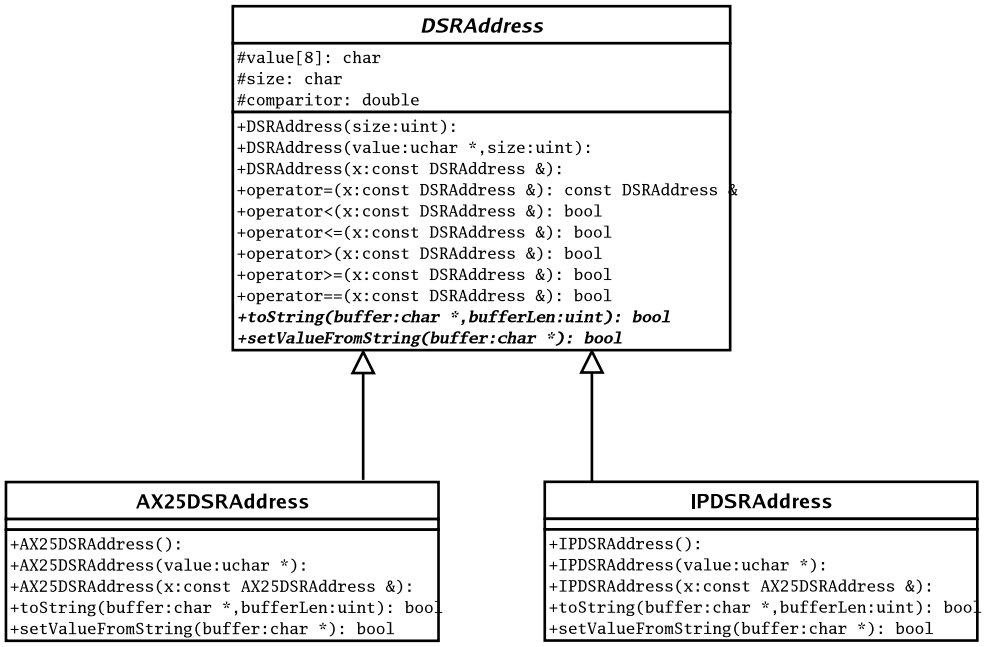

Figure 9: The IPDSRModule details.

address cannot exceed the bitwise size of the *double* datatype. The reason for this seemingly odd restriction is that, in order to allow extremely efficient comparisons of *DSRAddress* objects, *DSRAddress* contains a comparitor of type *double* that can be used to compare two addresses. The other reason for this restriction is that do avoid requiring the *DSRAddress* do perform dynamic memory allocation, the *value* field is an array with size capped at 8. Subclasses implement the *toString* and *setValueFromString* methods to allow nice printing of addresses when debugging or logging events and to populate addresses from the common, user friendly form for the given address type. The *DSRAddress* design and heirarchy is shown in figure 9. This is now described in detail:

value : Holds the actual address.

size : Holds the size of the address.

comparitor : Used to compare two *DSRAddress* object quickly. This is generated by basically copying bitwise *value* into the memory occupied by *comparitor* and then interpretting *comparitor* as a *double*.

$DSRAddress()$: The set of contructors for the $DSRAddress$ class.

operator <: Compares two $DSRAddress$ objects by comparing the *comparitor* values. Whether one address is 'larger' then another address conceptually has no significant but as long as the comparisons are consistent, they can be used as a key to a map - which will be extensively used in future sections.

toString : Must be overridden by subclasses to fill $buffer$ with a the human friendly version of the address.

setValueFromString : Must be overridden by subclasses to populate the *value* from the human readable address.

Figure 9 also shows the two subclasses of interest: $AX25DSRAddress$ and $IPDSRAddress$. Both of these implement the same set of constructors as $DSRAddress$ and implement the *toString* method to print addresses in the proper format for the given protocol. $AX25DSRAddress$ derives from $DSRAddress$ with the *size* automatically set to 7, $IPDSRAddress$ sets *size* to 4. Not that the $setValue$ functions required the passed address be in *encoded* form, not human readable form. The same is true for the constructors that accept a single $uchar*$ as parameter.

6.4 Route Cache

The route cache must store and allow the maintainence of all routes in a cache. Any subclass is responsible for providing mutual exclusion to data members as multiple threads may concurrently call any of the public method. The following methods must be provided by subclasses (as show in figure 10):

homeAddress : Contains the home address of the local node.

$flags$: Contains the flags for the *RouteCache*. Valid flags are as follows.

- $UNIDIRECTIONAL\_LINKS$: Set to indicate support for unidirectional links. Off indicates bidirection links can be assumed. Must be off in the current iteration.

RouteCache() : Constructs the route cache with the passed home address.

updateFromRReq : Updates the route cache with the information in the passed route request. An implementation may choose to perform the update or simply ignore the request. The route that could be added is the route from *src*, through all nodes in the *rreq*, to *homeAddress*. Possible return values:

- DSR_ROUTE_ADDED if the update results in a new route being added to the route cache.
- DSR_OK if the update was successful but did not causes any changes to the route cache.
- DSR_ERROR if an error was encountered during the update. *error* must be updated with a string indicating the error that occurred.

updateFromRRep : Updates the route cache with the information in the passed route reply. An implementation must add the route contained in the route reply to the route cache if the *homeAddress* is the destination address of the packet. If the destination of the route reply is not *homeAddress* an implementation may ignore the request. The route that may be added is the route from *dest*, through all nodes in the *rrep*. Possible return values:

- *DSR_ROUTE_ADDED* if the update results in a new route being added to the route cache.
- *DSR_OK* if the update was successful but did not causes any changes to the route cache.
- *DSR_ERROR* if an error was encountered during the update. *error* must be updated with a string indicating the error that occurred.

updateFromRErr : Updates the route cache by removing any paths containing the indicated link. An implementation must support this method (though not essential for iteration one since route errors are not used). Possible return values:

- *DSR_ROUTE_REMOVED* if the update results in a route being removed to the route cache.
- *DSR_OK* if the update was successful but did not causes any changes to the route cache.
- *DSR_ERROR* if an error was encountered during the update. *error* must be updated with a string indicating the error that occurred.

isPath : Returns the length of a path from source to destination or *DSR_NO_PATH* if no path between the source and destination exists in the route cache. If multiple paths exists it is up to the implementation to choose the 'best' path.

copyPath : Copy a path from the source to destination into the passed message. The *includeSrc* and *includeDest* flags indicate if the source and destination addresses respectively should be copied into the message as part of the path. I.e., if *includeSrc* is $false$ and *includeDest* is *true* the copied path would start with the first hop and end with *dest*. If multiple paths exists it is up to the implementation to choose the 'best' path. Possible return values:

- *DSR_NO_PATH* if no path between the *src* and *dest* exists in the route cache.
- *DSR_OVERFLOW* if the path is to large to copy into *msg*.
- Otherwise, returns the size (in bytes) of the path copied into *msg*.

xControl : Used to control the *RouteCache*. The valid opcodes are:

- *GET_UNIDIRECTIONAL_LINKS* : Return whether unidirectional links are being used. If this is not set, the *RouteCache* is free to assume all links are bidirectional.
- *SET_UNIDIRECTIONAL_LINKS* : Set to indicate support for unidirectional links. Off indicates bidirection links can be assumed. Must be off in the current iteration.

outputDebugInfo : Provides a virtual method to be used by subclasses to log debug information. Subclasses must implement this this method to output all route information in the cache to the passed file.

6.4.1 *PathCache* - A Simple Route Cache

The first iteration uses a very simple *RouteCache* subclass called *PathCache*. *PathCache* implements the route cache by each path having a single entry in a STL map (keyed by the source and destination addresses of the path). The *PathCache* contains the following:

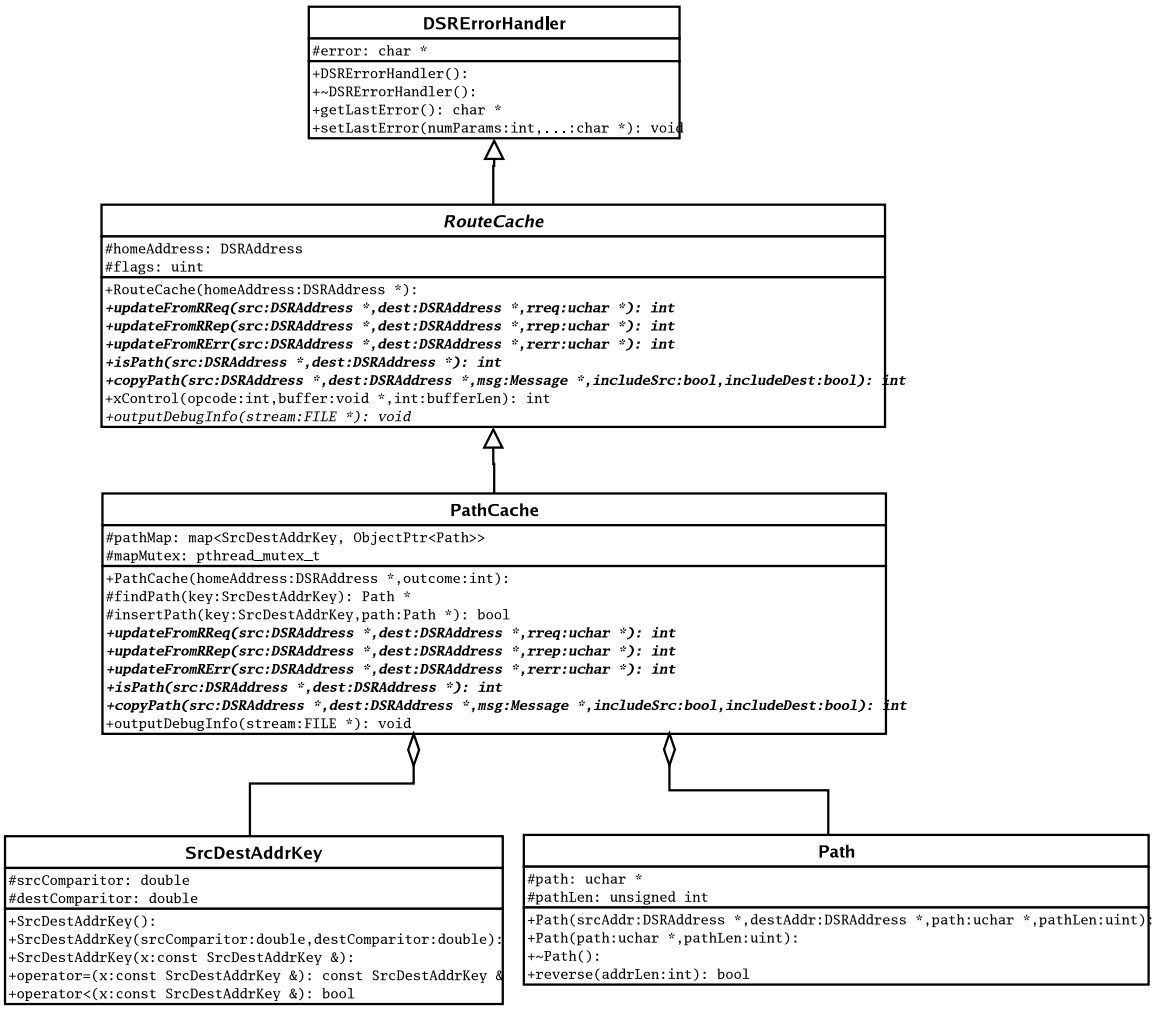

Figure 10: The *RouteCache* class and sample subclass.

pathMap : The STL map that contains all the known paths. The key is a *PathKey* object and the value is a *PathPtr* object.

mapMutex : Provides mutual exclusion to the *pathMap*. All methods that operate on *pathMap* must obtain this mutex before proceeding.

PathCache() : Constructs the route cache with the passed home address.

findPath : Searchs the *pathMap* for an entry with the indicated key. If a path is found it is returned, otherwise *NULL* is returned.

insertPath : Inserts the passed *path* into the *pathMap* under *key*. Returns whether or not the insertion was successful.

updateFromRReq : Updates the route reply from the information of the passed route reply. The route that can be added is the route from *dest* through all the addresses in *rrep*. The following three cases are possible:

- If *pathMap* contains no path between *dest* and the last address in the *rrep*, add this path to *pathMap*.
- If *pathMap* contains a shorter path between *dest* and the last address in the *rrep*, do nothing.
- If *pathMap* contains a longer path between *dest* and the last address in the *rrep*, replace it with this path.

updateFromRRep : If the *homeAddress* does not equal the destination address of the packet, it is ignored. If the *homeAddress* equals the destination address of the packet one of the following three cases occur:

- If *pathMap* contains no previous path between the source and destination, the path described by the route reply is added.
- If *pathMap* contains a previous path between the source and destination whose length (by hops) is shorter than the path in the route reply, ignore the route reply.
- If *pathMap* contains a previous path between the source and destination whose length (by hops) is not shorter than the path in the route reply, replace the existing entry by the path in the route reply.

updateFromRErr : Ignored.

isPath : Search *pathMap* for a path from *src* to *dest*. If one is found, return its length. If no path is found, and the $UNIDIRECTIONAL\_LINKS$ flags is off, search *pathMap* for a path from *dest* to *src*. If one is found return its length. Otherwise return NO_PATH.

copyPath : Search *pathMap* for a path from *src* to *dest*. If one is found, copy it into *msg*. If no path is found, and the $UNIDIRECTIONAL\_LINKS$ flags is off, search *pathMap* for a path from *dest* to *src*. If one is found copy the reverse path into *msg*. Otherwise return NO_PATH.

outputDebugInfo : Outputs each path in the cache to the passed file.

The *PathCache* implementation also requires the following two helper classes:

SrcDestAddrKey : Used as the key in the *PathCache* :: *pathMap*. Contains the comparitors for the source and destination addresses which allow a very efficient *operator* < implementation. These comparitors are derived from *DSRAddress* and are described in section 6.3. This class is also used by the *SimpleSendBuffer* as a map key (see section 6.5.1).

Path : Contains an actual path as a linear sequence of addresses starting with the source address and ending with the destination address. *pathLen* is the number of addresses (not bytes) in the *path*. The constructor accepts the source and destination addresses individually and the intermediate addresses as a linear array.

6.5 Send Buffer

The send buffer is used to store all packets that cannot be sent because a path to the destination is not known. When a new path is discovered and added to the route cache, any packets in the send buffer that can use this new path to reach their destination should be delivered. The send buffer is also required to enforce a timeout policy whereby packets can only reside in the send

buffer for a certain amount of time before being quietly discarded. Any subclass is responsible for ensuring mutual exclusion to any data members as multiple threads may call any methods concurrently. Figure 11 shows the *SendPath* abstract base class defined with the following attributes and methods:

dsrModule : Maintains a pointer to the *DSRModule* that contains this *sendBuffer*. This is required because when the *sendBuffer* sends packets it must use the *dsrModule* functionality.

sendBufferTimeout : The period (in seconds) that a packet may reside in the *SendBuffer* before being dropped.

SendBuffer : Create a new *SendBuffer* initializing it with the passed *DSRModule*.

queue : Add the passed packet to the list of outstanding messages. This must make a *copy* the *hdr* and *msg* and store these for use when a path is found.

routeAdded : Called by the *DSRModule* when a new route from *src* to *dest* is added to the *RouteCache*. Any packets in the *SendBuffer* destine for *dest* should be sent.

xControl : Used to control the *SendBuffer*. The following *opcode* values are valid:

- *GET_SENDBUFFER_TIMEOUT* : Gets the *sendBufferTimeout* value.
- *SET_SENDBUFFER_TIMEOUT* : Sets *sendBufferTimeout* to the indicated value.

outputDebugInfo : Should output a summary of the contents of the send buffer to *stream*. This will only ever be used for debug purposes.

6.5.1 *SimpleSendBuffer* - A Simple Send Buffer

Though named *SimpleSendBuffer*, this first iteration *SendBuffer* subclass is fairly complex and full featured. It allows for the queueing of unsent packets (associated by destination address) and has a thread responsible for periodically disposing of expired packets. *SimpleSendBuffer* contains the following attributes and methods:

expiryThread : The thread that periodically removes all expired entries from *sendBufferMap*.

mapMutex : Responsible for ensuring mutual exclusion to the *sendBufferMap*. Required since not only is the *expiryThread* periodically altering the *sendBufferMap*, but also several different threads may be attempting to send messages which must be placed in the *SendBuffer* at the same time.

sendBufferMap : Contains one entry per destination address that has packets queued for it. The key is a *SrcDestAddrKey* structure intialized with a source and destination address. Each value is a list (*SendBufferEntryList*) of all packets destine for the key destination that have not been sent.

SimpleSendBuffer() : Constructs the new *SimpleSendBuffer* initializing it with the passed *DSRModule*. Returns *DSR_OK* or *DSR_ERROR* in *outcome* to indicate success of object creation.

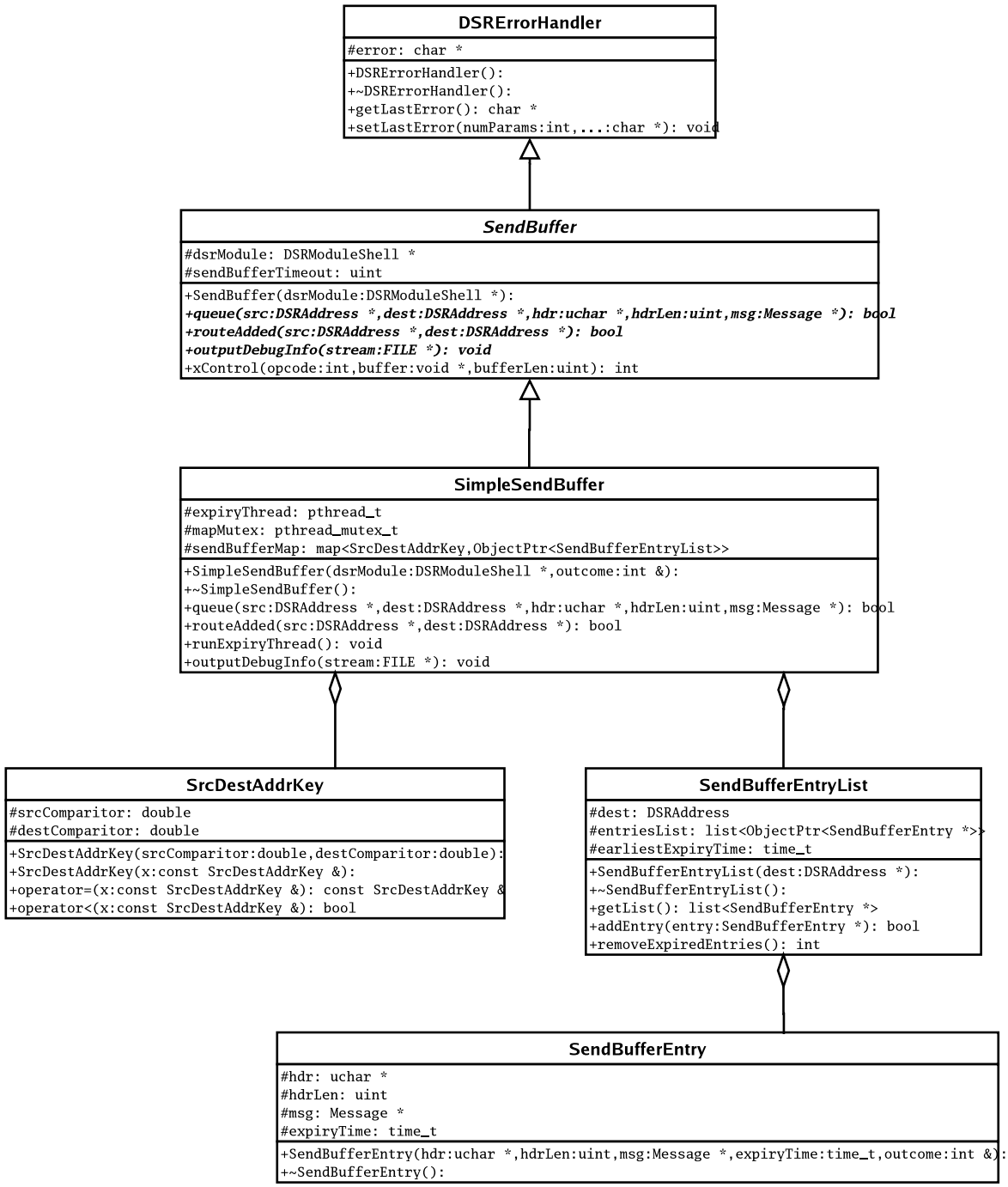

Figure 11: The $SendBuffer$ class and sample subclass.

$\sim SimpleSendBuffer()$: Destroys the $SimpleSendBuffer$ and deletes all outstanding packets in the $sendBufferMap$.

queue : Makes a new $SendBufferEntry$ structure that copies the *hdr* and *msg*. This new $SendBufferEntry$ is then added to the appropriate entry in the $sendBufferMap$ or a new $SendBufferEntryList$ is created if required.

routeAdded : Checks the $sendBufferMap$ for a $SendBufferEntryList$ for the passed destination. If one exists, all packets in that list are sent and the entry removed from the $sendBufferMap$.

runExpiryThread : Called by the *expiryThread*, this function endlessly loops removing any expired packets from the $sendBufferMap$ and then sleeping for $sendBufferTimeout$. Note that before doing a check of (and potentially modifying) the $sendBufferMap$, the $mapMutex$ must be acquired.

To accomplish its task $SimpleSendBuffer$ uses several helper classes. Figure 11 shows the relationship of the various $SimpleSendBuffer$ classes. Each of these classes are is now described:

$SrcDestAddrKey$: Used as the key to the $sendBufferMap$. Structure is as described in section 6.4.

$SendBufferEntryList$: Contains a list of all packets that have been queued for a particular destination. $earliestExpiryTime$ contains the system time at which the 'oldest' packet expires. If $earliestExpiryTime$ is later than the current system time, the $SimpleSendBuffer$ :: $runExpiryThread$ should call the $removeExpiredEntries$ to remove all expired entries and update the $earliestExpiryTime$. $addEntry$ should be called to add new entries to the list.

$SendBufferEntry$: Contains all the information for a single buffered packet. The constructor creates a new copy of the passed *hdr* and *msg* and the destructor is responsible for freeing this memory. $expiryTime$ records the system time at which the packet expires (i.e. should no longer be sent and should be removed from the $SendBuffer$).

There is a minor 'kludge' that is required here. Due to the fact that $SimpleSendBuffer$ is a template class, the $expiryThread$ main function (which is passed the $SimpleSendBuffer$ pointer and calls $SimpleSendBuffer$ :: $runExpiryThread$) cannot typecast the passed $void*$ to a $SimpleSendBuffer$ since it does not know the template parameter used to construct the class. To get around this, a non-template class, $ExpiryThreadBase$, was defined that contains the declaration of $runExpiryThread$ as a pure virtual method. $SimpleSendBuffer$ derives from $ExpiryThreadBase$ and the $expiryThread$ main function typecasts the passed $SimpleSendBuffer$ to a $ExpiryThreadBase$ to call the $SimpleSendBuffer$ :: $runExpiryThread$ method.

6.6 Route Request Table

The DSR draft (see [1]) describes a simple 'Route Request Table' data structure that is responsible for storing the information for both route requests forwarded by, and route requests initiated by, the home node. Any subclass is responsible for ensuring mutual exclusion to any data members as multiple threads may call any methods concurrently.The kind of information stored for these two types of route requests is very different as is the functionality of an implementation. Therefore, this implementation choses to break the route request table down into two separate classes: $ForwardedRReqTable$ and $InitiatedRReqTable$. Each of these data structures are now described in detail.

6.6.1 Forwarded Route Request Table

The *ForwardedRouteRequestTable* contains entries for route requests that have been forwarded by the home node. This does *not* include any route requests initiated by the home node. The main requirement for a subclass is to allow the quick determination of whether a given route request has already been seen. Figure 12 shows the *ForwardedRRequestTable* in detail; these details are now discussed:

requestTableIDs : As described in the draft ([1]) this is the maximum size of a queue of route request identifiers stored for any given initiating node.

ForwardedRReqTable : Initialize the instance with the passed maximum size for the queues.

setRequestTableIDs : Called to change the *requestTableIDs* variable. If the size if queues is being shrunk, some queues may have to have some entries dropped (the oldest) to meet the new maximum size requirement.

processRouteReq : Called whenever a received route request is processed by the *DSRModule*. This method must record the information for the route request in some kind of a queue associated with the *src* for the route request. This queue must have a maximum size of *requestTableIDs* and should be managed by some kind of FIFO algorithm. The following return values are possible:

- *DSR_RREQ_ALREADY_SEEN* : Indicates that this route request had already been seen (found in *src*'s queue). The old entry for this route request should have been moved to the end queue.
- *DSR_RREQ_NEW* : Indicates that this route request had not already been seen by the node. An entry for this route request will have been added to the end of *src*'s queue.
- *DSR_ERROR* : An error occurred processing the route request.

outputDebugInfo : Outputs the current contents of the route request table to the passed stream. This function should only be used for debug purposes.

xControl : Used to control the *ForwardedRReqTable*. The following *opcode* values are valid:

- *GET_REQUEST_TABLE_IDS* : Retrieve the maximum capacity of each queue in the table.
- *SET_REQUEST_TABLE_IDS* Sets *requestTableIDs* to the indicated value.

Figure 12 also shows the design of an implementation of a subclass of *ForwardRReqTable* - *SimpleForwardedRReqTable*. This implementation is described in detail below:

rreqMap : Contains one entry for each source address from which route requests have been received. The key to the map is a *SrcDestAddrKey* where the destination address is given comparitor value of 0. Each value is a *ForwardedRReqTableEntryList* object, each of which records all route requests (*ForwardedRReqTableEntry* objects) forwarded by the given node (in terms of target, id pairs) that have been received by the home node.

mapMutex : A mutex for *rreqMap*. Required as multiple threads may simultaneuosly call *ForwardedRReqTable* methods.

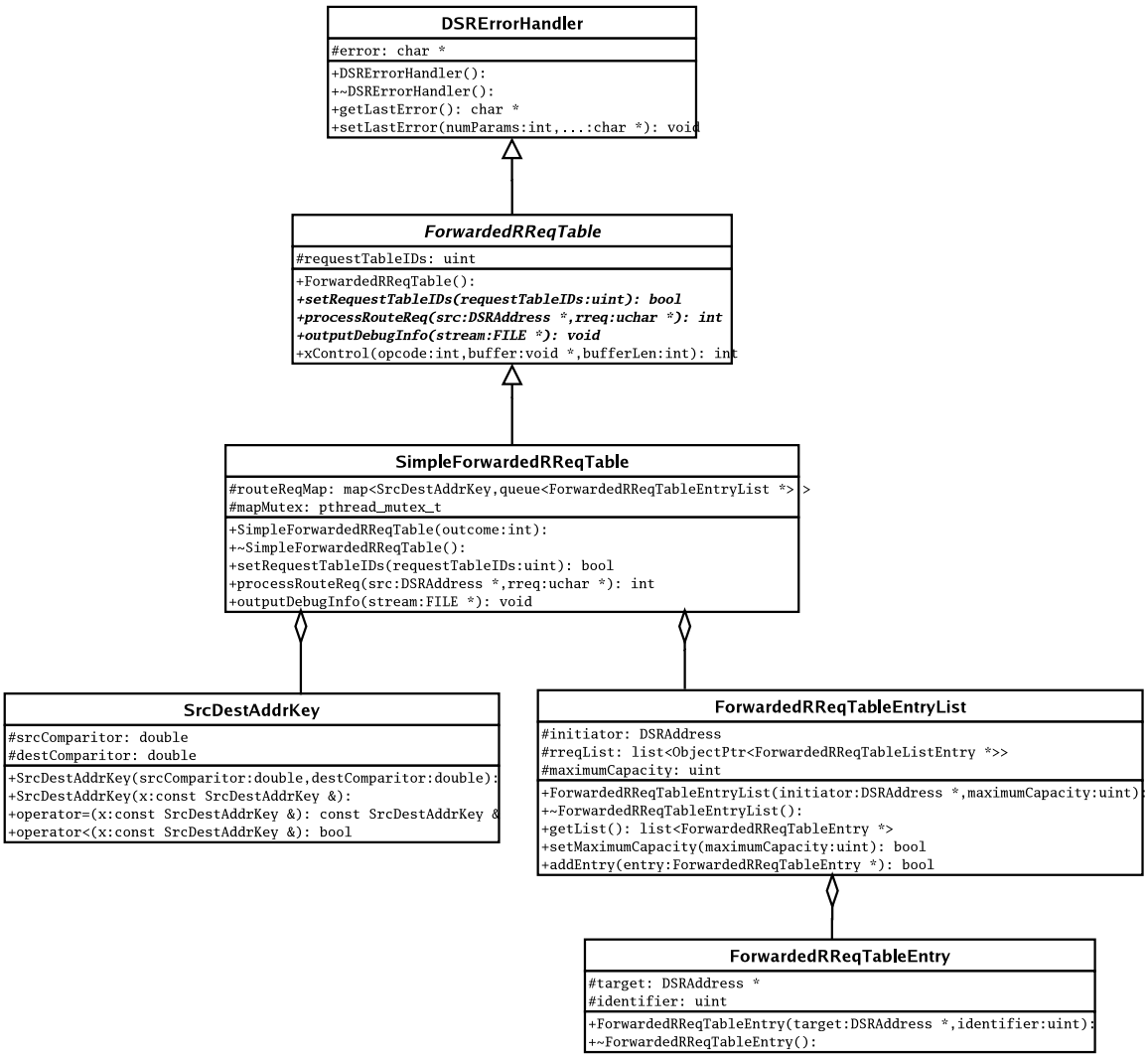

Figure 12: The *ForwardedRReqTable* class and sample subclass.

SimpleForwardedRReqTable : Initializes the *rreqMap* and *mapMutex*.

$\sim$ *SimpleForwardedRReqTable* : Release all the *ForwardedRReqTableEntryList*.

setRequestTableIDs : This must call the *setMaximumCapacity* on all lists in the map to ensure that upon return all lists have a maximum size of *requestTableIDs*.

processRouteReq : Add the passed route request to the *rreqMap*. An entry is either appended to an existing *ForwardedRReqTableEntryList* or a new *ForwardedRReqTableEntryList* for *src* is created if required.

outputDebugInfo : Outputs the current contents of the route request table to the passed stream. This function should only be used for debug purposes.

To accomplish its task *SimpleForwardedRReqTable* uses several helper classes. Figure 12 shows the relationship of the various *SimpleForwardedRReqTable* classes. Each of these classes are is now described:

SrcDestAddrKey : Used as the key to the *rreqMap*. Only the *src* field is used. Structure is as described in section 6.4.

ForwardedRReqTableEntryList : Contains a list of all route requests intiated by a given node that have been seen. *maximumCapacity* contains the largest size the list is allowed to be. List management is done using a FIFO algorithm. Adding an entry that is already in the list to the list results in the old entry being removed and the new entry placed at the back of the list.

ForwardedRReqTableEntry : Contains all the information for a single route request.

6.6.2 Initiated Route Request Table

The *InitiatedRReqTable* contains entries for route requests that have been initiated from this node. The *InitiatedRReqTable* is the class responsible for sending all route requests in the system - the *DSRSModule* simple adds routes it wants to find to the *InitiatedRReqTable*, and the *InitiatedRReqTable* is responsible for actually sending (and resending as necessary) of those requests. Figure 13 shows the *InitiatedRRequestTable* in detail; these details are now discussed:

dsrModule : The *DSRModule* to which this *InitiatedRReqTable* belongs. This is required so that the implementation can call the *DSRModule*'s *addRouteReqToMsg* and *rawBroadcastPush* methods as required to send route requests.

maxNumRReqRetransmissions : The maximum number of times a route request can be retransmitted before being removed from the table.

initialRReqTimeout : The number of milliseconds before a route request is to be resent the first time. I.e. a value of 500 means that the time between an initial route request being sent and the retransmission of that route request is 500 ms.

InitiatedRReqTable : Initialize the new object with the *DSRModule*. Returns *DSR_OK* or *DSR_ERROR* in *outcome* to indicate success of object creation.

addRouteReq : Add an entry for making route requests to *target*. This should trigger the sending of a route request to *target* as well as save the information to allow resending of route requests as appropriate. The following return values are possible:

- *DSR_RREQ_ALREADY_SEEN* : Indicates that an entry for *target* already exists in the table. No further action is necessary as the existing resending mechanisms will deal with the route requests to *target*.
- *DSR_RREQ_NEW* : Indicates that this *target* does not exist in the table. An entry for *target* have been added to the table and route requests will be sent in accordance with the exponential backoff scheme.
- *DSR_ERROR* : An error occurred.

removeRouteReq : Remove the route request table entry for the indicated *target*. This is called by the $DSRModule$ when a route reply to *target* is received. The following return values are possible:

- $DSR\_RREQ\_REMOVED$: The rreq for *target* was found and removed from the table.
- $DSR\_RREQ\_NOT\_FOUND$: The rreq for *target* was not found in the table.
- DSR_ERROR : An error occurred attempting to remove the entry from the table.

xControl : Used to control the $InitiatedRReqTable$. The following *opcode* values are valid:

- $GET\_RREQTABLE\_MAX\_RETRANS$: Get the maximum number of retransmissions allowed.
- $SET\_RREQTABLE\_MAX\_RETRANS$: Set the maximum number of retransmissions allowed.
- $GET\_RREQTABLE\_INITIAL\_TIMEOUT$: Get the time interval (in ms) between the first sending of a route request, and the first retransmission of that route request.
- $SET\_RREQTABLE\_INITIAL\_TIMEOUT$: Set the time interval (in ms) between the first sending of a route request, and the first retransmission of that route request.

outputDebugInfo : Outputs the current contents of the route request table to the passed stream. This function should only be used for debug purposes.

6.6.3 *SimpleInitiatedRReqTable*

Figure 13 also shows the design of an implementation of a subclass of $IntiatedRReqTable$ - $SimpleInitiatedRReqTable$. This class is *simple* in name only. It is actually a very full featured and efficient implementation of an initiated route request table. It support automatic resending of route requests using a exponential backoff algorithm. This implementation is described in detail below:

rreqMap : A map of *target* addresses to route request information. The key is a $SrcDestAddrKey$ with the source comparitor having a value of 0. The values in the map are objects of type $InitiatedRReqTableEntry$ which contain the number of times route requests have been sent for this target with no reply, the time of the last route request transmission, and the time the next transmission should occur at.

rreqSenderThread : The thread that is responsible for periodically scanning the *rrreqMap* and sending any route requests that are due to be sent. The thread only ever scans the *rreqMap* when there is a route request due to send or a new route request is added to the *rreqMap*.

mapMutex : Mutex to ensure mutual exclusion from the *rreqMap*. Any method that uses the *rreqMap* must first obtain this mutex as $SimpleInitiatedRReqTable$ methods may be called concurrently by many threads.

rreqAddedMutex : Used to protect the *rreqAdded* condition variable. This is required for calling *pthread_cond_timedwait*.

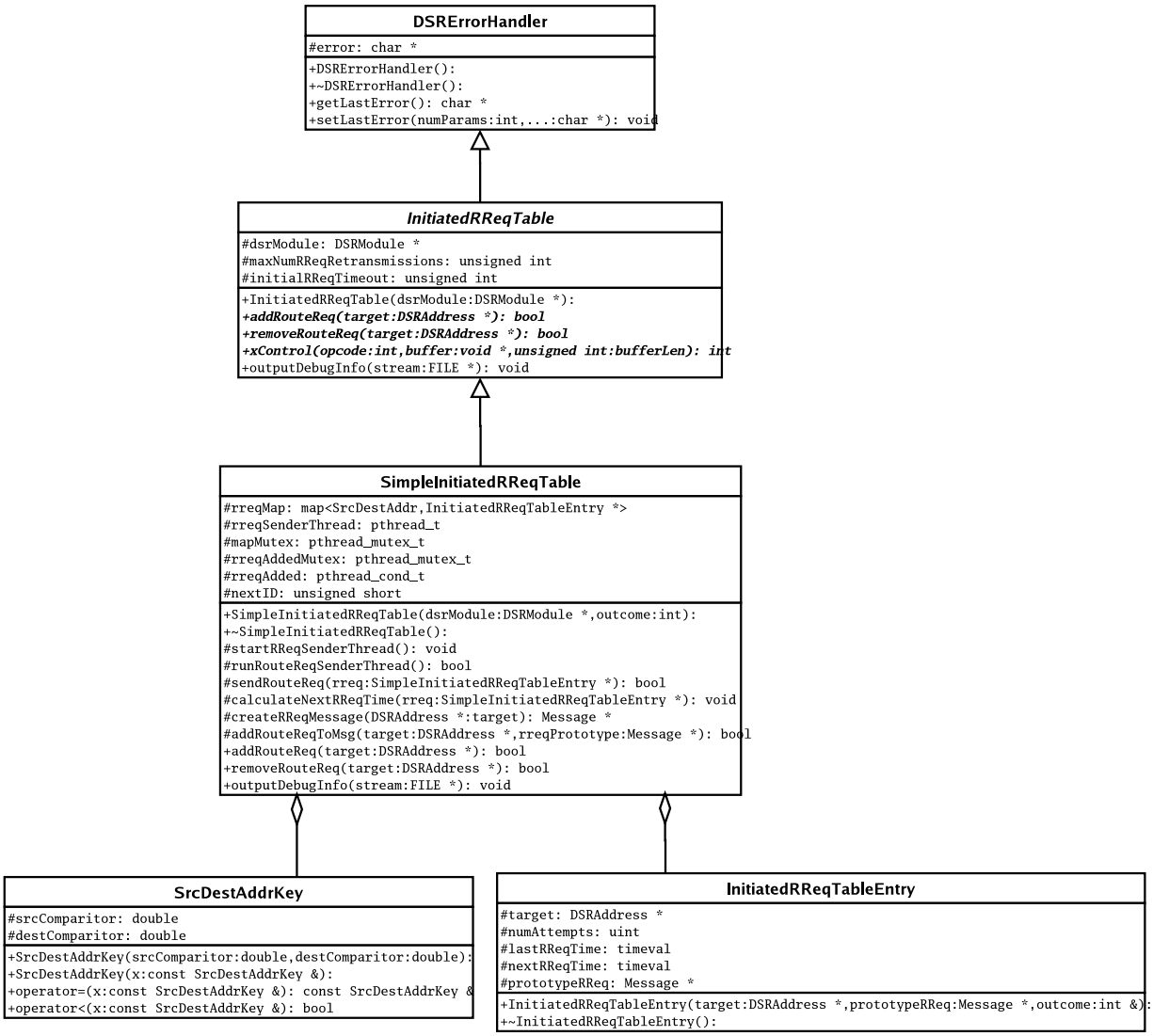

Figure 13: The *InitiatedRReqTable* class and sample subclass.

rreqAdded : A condition variable that is signalled whenever a new entry is added to the route request table.

nextID : Keeps track of the next available ID to be used for route requests.

SimpleInitiatedRReqTable : Initiate the object with the *DSRModule*. The *rreqSenderThread* should also be started. Returns *DSR_OK* or *DSR_ERROR* in *outcome* to indicate success of object creation.

$\sim$ *SimpleInitiatedRReqTable* : Destroy the object (stopping the *rreqSenderThread*).

addRouteReq : Called by the *DSRModule* when a path to *target* is needed. If there is already an entry in *rreqMap* for *target* nothing needs to be done. If there is no entry in the

rreqMap, a new entry should be added, a route request for this *target* should be immediately sent, and the time of the next *rreq* should be calculated so the *rreqSenderThread* knows when to resend the route request.

removeRouteRequest : Delete the entry in *rreqMap* for *target*.

startRReqSenderThread : Start the *senderThread*.

runRouteReqSenderThread : The main method of the *rreqSenderThread* which resends route requests as necessary. The main loop of this method is only executed when there is a route request requiring retransmission or a new route request has been added to the table.

sendRouteReq : Called to send a route request to the *target* indicated by *rreq*. This should also call *calculateNextRReqTime* to calculate and update the time of the next route request transmission for the *target*.

calculateNextRReqTime : Calculates and updates the passed *InitiatedRReqTableEntry* with the next time it is eligible to send a route request for the target.

createRReqMessage : Create a prototype route request message for the passed *target*. Whenever a route request for *target* needs to be sent, the prototype will be used.

To accomplish its task *SimpleForwardedRReqTable* uses several helper classes. Figure 13 shows the relationship of the various *SimpleInitiatedRReqTable* classes. Each of these classes are is now described:

SrcDestAddrKey : Used as the key to the *sendBufferMap*. Structure is as described in section 6.4.

InitiatedRReqTableEntry : Contains all the information required for a single address that has outstanding route requests. Last and next transmission times are stored with microsecond percision. A prototype route request to the *target* is also stored.

The same technique described in section 6.5.1 is used to allow the calling of the *runRouteReqSenderThread* method on template class *SimpleInitiatedRReqTable* from a nontemplate main thread function.

6.7 Gratuitous Route Reply Table

This data structure is not applicable to the current iteration and will be defined as an empty class *GratRRepTable*.

6.8 Retransmission Buffer

This data structure is not applicable to the current iteration and will be defined as an empty class *RetransBuffer*.

7 Putting it All Together

This document has described the $DSRModule$ as a class that uses several helper classes to assist it. In section 3.1 a simple, abstracted way of creating an $AX25DSRModule$ was presented with the note that some details were deferred for later discussion. The details are the creation and binding of the various data structures associated with a $DSRModule$. There are two main topics that must be looked at to see how a functional $DSRModule$ can be constructed: the generation of the helper classes and the construction of the $DSRModule$.

7.1 Generating the Helper Classes

In order to allow the helper classes to be generic, generative programming is used to construct some of these classes. This applies to the $RouteCache$, $SendBuffer$, and $RouteRequestTable$. To detail how this works the process for generating the $RouteCache$ subclass $PathCache$ will be detailed. Generating the other helper classes uses a similar methodology.

The following files are used to generate the $PathCache$:

PathCacheGenerator.hpp: Defines the generator of the $PathCache$. The generator has two parameters:

- *Message:* The class used to represent messages. This class must support the *push*, *peek* etc. method supported by the $SingleBufferMessage$ class.
- *DSRAddressClass:* The class used to represent addresses. This class must derive from $DSRAddress$. This information is used to both define the base $RouteCache$ and the subclass $PathCache$.

PathCacheConfig.hpp: Provided by the user, this file simply typedefs the generation of a $PathCache$ of the desired type by using the generate. A simple example would look like this:

```
typedef PATH_CACHE_GENERATOR<SingleBuffMessage,AX25DSRAddress>::RET
   AX25PathCache;
```

Now, a user can simply include $PathCacheConfig.hpp$ and define an object of type $AX25PathCache$.

PathCache.hpp: The $PathCache$ class is now a template class that accepts a single parameter: the generator (as defined in $PathCacheGenerator.hpp$). $PathCache$ is then defined to derive from the $RouteCacheBase$ (generated and stored in the passed generator) and also use the appropriate class types ($DSRAddressClass$ and $Message$).

RouteCache.hpp: The $RouteCache.hpp$ class is now a template class that accepts a single parameter: a $Config$ structure that contains the $DSRAddressClass$ and the $Message$ class.

7.2 Constructing the $DSRModule$

To construct the DSRModule (once all the helper classes have been generated the user simply calls set methods (i.e. $setRouteCache$, $setSendBuffer$, etc) passing in the constructed helper objects. The example $AX25DSRModule$ creation now looks like this (error handling omitted for clarity):

```
1  AX25Protocol *  ax25 = new AX25Protocol(device, result);
2  AX25DSRModule * ax25DSRModule = new AX25DSRModule(homeAddress, ax25);
3  ax25DSRModule->setRouteCache(new AX25PathCache(homeAddress));
4  // ... set other helper classes ...
5  ax25DSRModule->setRetransBuffer(NULL); // Since not in this iteration.
6  ax25DSRModule->setGratRRepBuffer(NULL); // Since not in this iteration.
7  ax25->xControl(SET_DSR_MODULE, ax25DSRModule, 0);
```

References

[1] Johnson, David B., et. al. *The Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR)*, IETF Draft (November 2001).

[2] Beech, Nielson, and Taylor. *AX.25 Link Access Protocol for Amateur Packet Radio*, Version 2.2. Tucson Amateur Radio Corporation, 1997.

[3] Campbell, Greg. *DSR Implementation Preliminary Design*, January 2002.

[4] Barbeau, Michel. *An Implementation of DSR in IPv4*, December 2001.