2006

HF-DSR: dynamic source routing for high frequency radio networks

Michael Stringer

Department of Computer Science

Rochester Institute of Technology

MS Project

HF-DSR

Revision: 0.4

Date: March 20, 2006

Author: Michael Stringer (mstringe@gmail.com)

Approvals:

Chair	Alan Kaminsky	Date
Reader	Hans-Peter Bischof	Date
Observer	Stanisław Radziszowski	Date

1 Abstract

HF-DSR is an ad-hoc routing protocol designed to operate efficiently over HF radio networks. Ad-hoc routing protocols allow networks to provide dynamic routing between endpoints. In contrast, static routing schemes require networks to be configured with potential routes in advance. As such, ad-hoc routing mechanisms can compensate for unanticipated factors such as RF interference or node mobility.

HF-DSR is largely based on DSR [Johnson+, 2001], a level 3 ad-hoc networking protocol that emits network control information on demand. HF-DSR incorporates portions of DSR which minimize the quantity of control information transmitted. In this manner, as much network bandwidth as possible is conserved for user-initiated data transfers.

During the project associated with this report, the following activities occurred: An implementation of HF-DSR was developed to operate over the NATO standard STANAG 5066 data transfer protocol. A four node network was assembled using Microsoft Windows PCs, HF modems, HF radios, synchronous RS-232 interfaces, and Harris RF-6750W Wireless Gateways. Two different network topologies were constructed using the four nodes. Finally, HF-DSR route discovery and file transfer was exercised on both network topologies.

1.1.1.1Background

1.1.1.2Radio Communications

HF radio's primary benefit is the ability of its characteristic low frequency radio waves to refract off of the Earth's ionosphere. In this manner, the radio waves progress from the transmitter to the ionosphere, and diffract from the ionosphere back to the surface. This behavior is called a “hop”. The distance achievable by a single hop provides “Beyond Line-of-Sight” (BLOS) communication, and multiple hops are possible. Higher frequency waves cannot diffract from the atmosphere to the Earth's surface. As a result, radio waves above 30mhz are limited to line-of-sight communications.

HF radio communications that utilize one or more hops to reach a receiver are called “sky wave” communications. Sky wave communications are frequently utilized by ground-based facilities or mobile organizations for whom more favorable communications media are unavailable.

HF radio waves also propagate well over a continuous body of water. This type of propagation does not involve any atmospheric hops. Instead, the waves follow the surface of the water due to different refractive indices in the water and the atmosphere. In this manner, BLOS communication is achieved without sky wave propagation. Communication utilizing this phenomenon is referred to as “ground wave” communication.

Ground wave communication provides consistently high-quality signal strength to receivers within range. As such, it is heavily utilized within navies throughout the world.

In contrast to its benefits, HF radio has several disadvantages when compared with other wireless communication mechanisms. HF radio channels are 3khz wide, which limits practical data rates to 9600 bits per second (bps). Communications reliability can vary or fail completely due to interference, time of day, and type of propagation being used. Additionally, HF channel bandwidth is smaller than ultra-high frequency (UHF) channel bandwidth due to the relatively small bandwidth of the entire HF band. The HF band constitutes the range between 3mhz and 30mhz. The UHF band is composed of the frequencies between 300mhz and 3000mhz and commonly uses 25khz channels.

1.1.1.3Data Communication Over HF Radio

Data communication across an HF radio channel is accomplished by means of an HF modem. HF modems convert data into audio for radio transmission, and convert received audio into data. Due to the variability of received signal quality, HF modems implement several techniques to recover from unfavorable attempts to demodulate data. The specific techniques available depend on the HF waveform being used.

Most HF waveforms implement some level of forward error correction (FEC). Data loss is a certainty on any realistic HF circuit and FEC allows certain quantities of consecutive bit errors to be automatically corrected. The ratio of FEC overhead bits to user-submitted bits is typically so high that the additional overhead is presented to users as lower data rates. In these cases, the modem transmits at the same relatively high data rate while drastically increasing the radio of FEC overhead to user-submitted data. For example, the MIL-STD-188-110A modem waveform always transmits at 4800bps and users can change their apparent bit rate from 75bps to 4800bps.

Most HF waveforms implement data interleaving. The purpose of interleaving is to recover from a stream of consecutive bit errors too numerous for FEC to handle on its own. Interleaving operates by delineating transmission length into time periods. The amount of data encoded during each time period is called the "interleaver block". HF modems scatter consecutive data submitted within each interleaver block before transmission. The receiver "un-scatters" incoming data after de-modulating it from the received audio. In this manner, consecutive errors applied to the interleaved data will not necessarily map to consecutive errors in non-interleaved data. Minimizing consecutive bit errors in non-interleaved data makes it more likely that the receiving modem can correct them using FEC.

Data interleaving causes the transmitting modem to modulate audio for between one and two interleaver block lengths longer than it would if interleaving were not being used. This increases the characteristic latency of HF channels significantly.

1.1.1.3.1.1MIL-STD-188-110A

One waveform of interest is MIL-STD-188-110A, commonly referred to as "110A". It is also called the "Serial Tone" waveform due to its characteristic single tone audio. 110A specifies several features that simplify data transfer. It is an autobauding waveform, which means that modems implementing 110A need not configure incoming data rate and interleaver settings. In contrast, non-autobauding waveforms require modems to be configured to specific settings in order to successfully demodulate audio. 110A has three interleaver settings allowing interleaver blocks of 0, 0.6, or 4.8 seconds at all data rates. Additionally, 110A encodes a sentinel token at the end of every transmission to inform receivers that the reception is complete. Waveforms that do not encode trailing sentinels rely on receiving modems to determine the end of a reception based on unreliable heuristics such as signal loss. Finally, 110A has the capability to demodulate data mid-stream. It does not require that the beginning of the transmission be received, which distinguishes it from other waveforms.

1.1.1.4Data Link Protocols over HF Radio

1.1.1.4.1.1Channel Access

Resolving contention between multiple nodes on an HF channel can be particularly problematic due to the high latency associated with the medium. If a node on an HF network transmits data at an interleaver setting that causes an interleaver delay of 2n seconds, then receiving nodes will not detect the reception until n seconds have elapsed. As a result, there will be contention on an HF channel if multiple nodes begin transmitting within half of an interleaver delay of one another. These characteristics must be accommodated by ensuring that multiple stations with simultaneous outgoing data defer transmission until it is their turn to send. This can be accomplished with slot-based or token passing schemes.

"Time Division Multiple Access" (TDMA) [TDMA] is able to overcome this problem at the expense of bandwidth and required control information. TDMA operates by providing each node a non-overlapping time slice that allows it to begin a transmission uninterrupted. While this allows for each node in the network to claim the channel without interference from other nodes, time slices remain allocated for all nodes regardless of whether they have data to transmit. As a result, channel throughput decreases proportionally to the number of nodes in the network. Additionally, time slices must be dynamically assigned to nodes entering the network and unassigned from nodes leaving the network. Additional control information is required to perform this negotiation, which further decreases throughput and introduces a point of failure.

Token passing schemes utilize a logical "token" that only one node in the network can hold at once. Only the node holding the token is allowed to transmit. All other nodes in the network are restricted from transmitting. In this manner, no two network nodes can transmit simultaneously, which creates an effective channel access mechanism. Token transfer from one holder to another is accomplished by sending and receiving over-the-air commands and responses.

Although token passing schemes provide benefit, they are subject to the "lost token" problem. The "lost token" problem occurs when the token cannot be transferred from the current holder to the next holder. Regardless of how robust the token passing command sequence is, token loss is possible due to changing network connectivity. Additionally, handling token loss on ad-hoc networks becomes more complex due to changing network membership.

"Collision Avoidance with Control Handshaking" (CACH) [Holcomb+, 2003] operates by waiting until an ongoing transmission ends, and then randomly choosing a time slice to begin sending its data. The time slices are long enough that traffic started at the beginning of time slice N will be detected by receivers before the beginning of time slice N+1. Nodes waiting until a time slice to transmit their data will postpone

their transmission if a reception is detected beforehand. In this manner, the current transmission is unlikely to be interfered with.

An advantage of CACH is that minimal control information is transmitted between nodes, which is a critical consideration in low-bandwidth networks. Additionally, the presence of higher-level protocol traffic (such as HF-DSR) causes nodes to switch into a monitoring state. Monitoring nodes defer their outgoing data until such time as monitored data ceases. Finally, the ability of multiple nodes to queue data allows for higher network throughput than other schemes such as token passing, which let only one network member transmit at once. CACH networks allow multiple nodes to transmit simultaneously as long as they are unable to receive one another's transmissions.

One disadvantage of this scheme is transmitting during the same time slice may interfere with one another. These collisions must be resolved by higher-level protocols. [Holcomb+, 2003]

Some data link protocols periodically encode the time remaining in each transmission along with data and control information. This feature greatly reduces the likelihood of over-the-air collisions. If a receiver starts transmitting before a received end-of-transmission (EOT) timer expires, there will be a collision. Therefore, receivers start timers corresponding to received values of the EOT timer and only transmit when the EOT timer expires.

1.1.1.4.1.2Automatic Repeat Request

Data loss is not completely mitigated by interleaving and FEC. A common feature to overcome any remaining data loss is “Automatic Repeat Request” (ARQ). ARQ allows receivers to detect data blocks in error and inform senders which blocks require retransmission. Receivers utilize cyclic redundancy checks (CRC) or similar mechanisms to detect blocks in error. Next, data acknowledgments (ACKs) are constructed with information concerning which blocks were in error. At this point, receivers send the ACK to the sender. Finally, the sender resends missing data along with previously unsent data (if possible).

Data link protocols frequently impose constraints to reduce ACK complexity and length. Sliding transmit and receive windows (Tx and Rx windows) limit the number of unacknowledged data blocks in the communication circuit. Sliding window protocols operate by assigning sequence numbers to individual data blocks. The Tx and Rx windows are both initialized to size 0, and have a maximum sequence number range. Every data block transmitted by the sender increases the Tx window size by one. The maximum size of the Tx window cannot be exceeded, so the amount of outstanding data is limited. In contrast, the Rx window remains at size 0 as long as no error blocks are received. In this case, the receiver constructs an ACK with the next expected sequence number N. If an errored block is received, the Rx window grows in size with every subsequent block received. In this case, the ACK contains the next first sequence number in error N along with an boolean vector representing whether sequence numbers (N + vector index) have been successfully received.

Upon receiving the ACK, the sender removes all entries from the Tx window representing data blocks with lower sequence numbers than N. If a boolean vector is available, every vector index I with value “true” causes the Tx window to remove the entry for sequence number (N + I + 1). Next, the sender can resend data blocks for which there are entries in the Tx window. If processing the ACK caused the range of sequence numbers in the Tx window to be less than the maximum, the sender may send additional data blocks until the Tx window reaches its maximum range.

Some data link protocols include a supplemental mode that omits ARQ (non-ARQ). In lieu of receiver-directed repeat requests, the sender will frequently encode over-the-air data multiple times. This mechanism can potentially overcome errored blocks.

1.1.1.4.1.3Addressing

Most data link protocols contain some form of addressing that identifies the sender and one or more receivers. In this manner, only valid receivers need to process incoming data. Analogously, only valid

receivers are required to transmit responses. Channel acquisition contention is correspondingly reduced. Additionally, potential senders may monitor over-the-air traffic involving other senders and delay their transmissions, reducing channel collisions.

Data link protocols handle communication between source and destination addresses that are directly reachable. Transfers spanning one or more intermediate node are managed by higher-layer network layer protocols.

1.1.1.4.1.4STANAG 5066

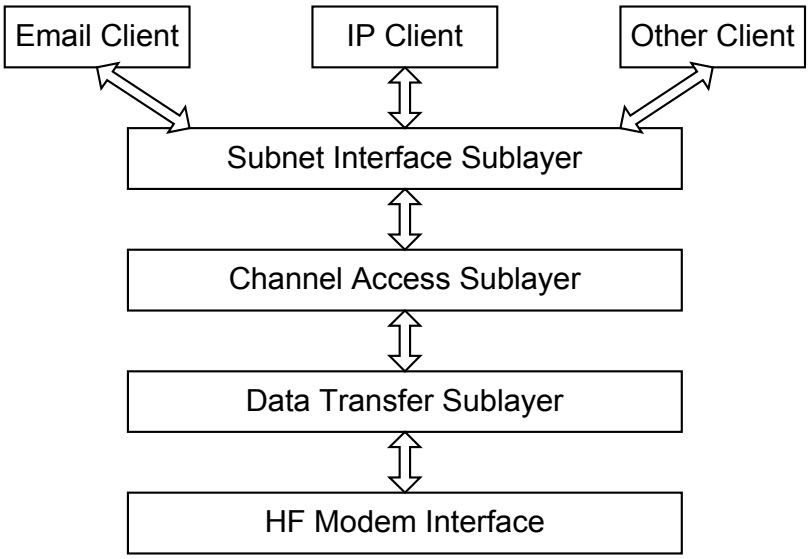

Figure 1: STANAG 5066 Architecture

NATO Standard Agreement 5066 (STANAG 5066) [STANAG 5066] specifies a data link protocol that promotes reliability and maximizes throughput. STANAG 5066 specifies a sliding window ARQ scheme that allows data transmissions up to two minutes. In this manner, interleaving delays caused by a larger quantity of shorter data transmissions are mitigated. Effective bi-directional data transfer is facilitated by sending ACKs and data in the same transmission. Collision avoidance is aided by end of transmission timer values encoded periodically throughout individual transmissions. Non-ARQ mode is available with configurable levels of redundancy for cases in which ACKs are impossible or undesirable.

STANAG 5066 contains three primary subsystems. The DTS (Data Transfer Sublayer) contains the ARQ engine and is primarily responsible for assembling complete C_PDUs (Channel Access Sublayer Protocol Data Units) from incoming D_PDUs (Data Transfer Sublayer Protocol Data Units). Conversely, outgoing C_PDUs are segmented into one or more D_PDUs. The CAS (Channel Access Sublayer) implements basic channel access and is responsible for establishing, breaking, and monitoring node-to-node connections throughout the network. The SIS (Subnetwork Interface Sublayer) provides an interface for external clients to communicate with.

The DTS interfaces with a synchronous RS-232 interface by encoding data for transmission and decoding incoming data. The DTS utilizes three major approaches in order to facilitate effective communication over variable channel conditions. First, signal-to-noise ratio (SNR) is sampled from the modem during receptions. The SNR measurements map to preferred incoming data rates and interleaver settings. The resulting settings are suggested to the peer in a subsequent transmission.

The second approach used by the DTS to mitigate channel conditions is to control the modem's data rate and interleaver settings. Two major criteria for adjusting the modem are outgoing packet success rate and remote suggestions from the peer. These factors are combined to select improved modem settings for the immediate future.

The third approach is to vary the size of encoded D_PDUs. Each D_PDU is independently verified by cyclic redundancy check. Therefore, smaller D_PDUs are statistically more likely to pass CRC validation on a channel with constant bit error rate (BER). On the other hand, smaller packets reduce channel efficiency by decreasing the ratio of user-submitted data to packet overhead. Determining D_PDU sizes requires balancing these factors to achieve good results at a given time.

Channel access implemented by the CAS is moderately effective when multiple STANAG 5066 nodes are operating on the same radio channel. Data exchanges between pairs of nodes are preceded by CAS level negotiation called physical link establishment. Analogously, physical links are broken by CAS level negotiation following data exchanges completion. STANAG 5066 nodes external to a physical link can monitor the link's existence and delay outgoing traffic until the link has broken.

External clients communicate with the SIS with data structures called S_PRIMITIVEs. S_PRIMITIVEs are exchanged between the SIS and clients by a TCP connection on the well-known port 5066 [IANA]. Three main categories of S_PRIMITIVEs are available. The first category allows clients to bind and unbind. This allows the SIS to accept or reject new client connections based on internal criteria. The second category allows for data submission, reception and a selectable amount of delivery confirmation. Finally, the third category involves the explicit creation of physical links by clients. This category is not heavily exercised by practical STANAG 5066 clients..

Client types specified by STANAG 5066 include two email clients and an IP client. Other clients include the STANAG 4406E formatted messaging system and a simple chat program.

1.1.1.4.1.5MIL-STD-188-141A

STANAG 5066 lacks the capability to automatically change radio frequency if its current channel no longer propagates. A second data link protocol must be used for this purpose. One such protocol is MIL-STD-188-141A Automatic Link Establishment (ALE). ALE has the ability to select an operational channel from one or more channel groups. This is accomplished by passive scan through several channels every second. During the scan, ALE listens for a node attempting to establish a link on a particular channel. Following negotiation between the nodes, they remain on the same channel and are considered connected. Additional ALE negotiation occurs when a connected node abandons the link and returns to scanning through channels.

ALE allows several types of links. “Individual” calls attempt to place an initiator and selected receiver on the same channel. “Net” calls arrange as many nodes from a pre-configured set as possible on the same channel. “Any” calls attempts to link with one node from a group. Finally, “All” calls link with as many nodes as can detect the call attempt.

ALE has extremely limited data payload transmission capability. As such, it is typically used in conjunction with other data link protocols, such as STANAG 5066.

1.1.1.5Networking Protocols over HF Radio

HF radio networking interprets a collection of radio nodes that can receive transmissions from one another as a subnetwork. This is analogous to Ethernet nodes that can receive packets from one another without using a router being an Ethernet subnetwork. One major distinction between the network types is that Ethernet routers are reachable from all nodes in the subnetwork. In contrast, radio routers vary between being reachable or not reachable over time. As a result, operational radio networks require redundant routing capability throughout their nodes to maximize the likelihood that at least one router is reachable.

In general, the quantity of over-the-air data associated with a routing protocol varies inversely to the time required to successfully communicate from a source to a destination. At one extreme is a routing protocol that statically configures routes from all source nodes to all destination nodes. This static protocol needs no

data transmissions to generate routes because complete information is available at every node. However, statically configured routes are unlikely to reflect the reality of network conditions at a given time. This static protocol would submit user data along a pre-configured route. Upon failure, another route would be attempted. This process would continue until the user data reached its destination or until all routes were exhausted. The advantage of this protocol is that network traffic consists entirely of user data. However, discovering a working route could take time proportional to the number of node combinations in the network.

An improvement to the static routing protocol is a dynamic protocol that relies on searching the network for a destination. In this case, a source node would broadcast user data to all adjacent nodes. Non-destination nodes that receive some user data would re-broadcast it following the first reception. An advantage of this protocol is that user data can reach its destination along a newly-discovered path through the network. However, there are several disadvantages. One problem is that the entire network gets searched even if the final destination is found early. Additionally, channel contention becomes severe as the number of nodes in the network increase. Finally, the search procedure must be invoked for all user data submitted.

Neither of the abovementioned routing protocols would both perform well in practice. Actual networking protocols attempt to use a small quantity of transmitted data to discover optimal network paths. This is accomplished using different approaches.

1.1.1.5.1.1Link State Routing

One class of routing algorithms is called “link state routing”. Link state routing protocols operate by having each node maintain a partial or total network topology. Periodically, each node in the network communicates its stored topology to all nodes within range. The topology updates allow receivers to update their route calculations for all destinations. Since all all routes are assembled during steady state operation, submitted user data may proceed immediately along a pre-discovered path.

A disadvantage of link state routing protocols is that they do not scale well over HF networks. Increasing the number of nodes in a network causes the number of topology notifications to increase proportionally. As a result, the network bandwidth available for user data transfer is reduced. Simultaneous transmissions can be avoided by staggering topology notification schedules for all nodes. Additionally, increasing the delay between topology notifications can offset bandwidth reduction at the expense of maximum route age.

1.1.1.5.1.2OLSR

OLSR (Optimized Link State Routing) [Clausen+, 2003] is a link state routing protocol designed to operate over wireless ad-hoc networks . OLSR describes a network topology that is composed of node clusters. These clusters resemble points on a circle with bi-directional links to a “multipoint relay” (MPR) node at the center. OLSR organizes nodes into clusters and selects at least one MPR per cluster. MPRs are responsible for communicating and re-transmitting user data and link state information between clusters. Nodes that are not MPRs do not forward information beyond their clusters. Depending on network topology, the network can grow larger while the number of MPRs remains constant. In this manner, OLSR can be highly scalable.

Any node running OLSR may be selected as an MPR, although ideal MPRs are subject to certain criteria. One requirement is that MPRs need bi-directional connections with as many client nodes as possible. In this manner, MPRs can represent more nodes to the larger network. An additional factor is that nodes intending to be MPRs must indicate their willingness to do so. As a result, low bandwidth or low power nodes are spared the extra responsibility of being an MPR.

Nodes within a cluster periodically send “HELLO” messages to other nodes within a one hop distance. In this manner, bi-directional connections with neighbor nodes are discovered. A subset of these nodes may be designated as MPRs. Multiple MPRs may be selected to improve reliability at the expense of additional network traffic.

MPRs communicate their local topologies by using "Topology Control" (TC) messages. TC messages are not transmitted by non-MPR nodes. TC messages contain information on all nodes that the originating MPR represents to the network. Peer MPRs rebroadcast received TC messages in order to propagate topology information throughout the network.

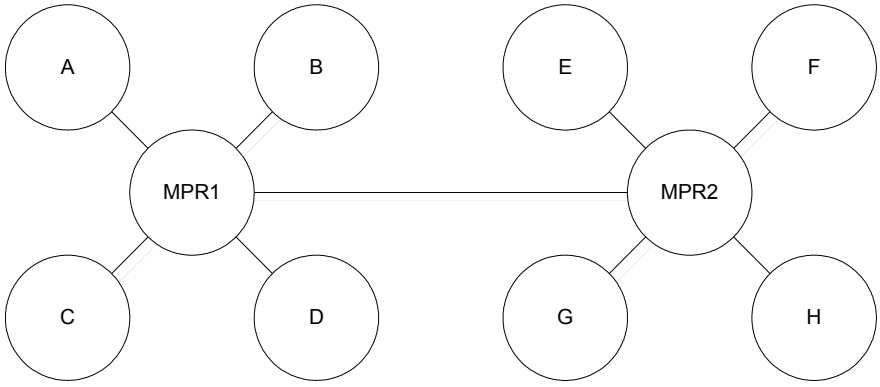

Figure 2: Two OLSR clusters

OLSR operates most effectively over line-of-sight wireless media because node connectivity worsens as distance between nodes increases. As a result, the one hop limitation for HELLO messages ensures that node clusters are geographically proximate. In this situation, HELLO messages do not affect global network bandwidth. Increases in global network traffic do occur relative to the quantity of MPRs within the network. This is due to TC message transmission and re-transmission quantity increasing linearly to MPR quantity. MPR quantity increases with the spacial coverage in a line-of-sight network. As a result, global network traffic increases with geographic network size, not network node density. However, a consistent increase in node density throughout the network would affect the network in a similar manner.

OLSR would perform much differently on an HF radio network. HELLO messages would propagate much farther than they would using a line-of-sight network. This would tend to organize a large HF network into a very small quantity of clusters. TC messages would be a minor contributor to global traffic because of the low quantity of MPRs. Conversely, HELLO messages would affect global network traffic to a much greater extent due to improved single hop propagation. As a result, global network traffic would rise in proportion to the number of nodes in the network. This contrasts with scalability characteristics over line-of-sight networks.

1.1.1.5.1.3On Demand Routing

"On-demand routing" is a type of routing algorithm that differs from link state routing. On-demand routing does not discover routes while the network is in a steady state. Instead, route discovery is initiated on-demand by a source node that requires a route to a particular destination. This contrasts with link state routing, which discovers routes between nodes during steady state operation.

1.1.1.5.1.4DSR

DSR (Dynamic Source Routing) [Johnson+, 2001] is an on-demand ad-hoc routing protocol. DSR incorporates two major tasks: Route discovery and route maintenance. Route discovery disseminates node-to-node routes throughout the network. Route maintenance determines whether or not existing routes are still valid.

Route discovery operates by broadcasting a "route request" to determine a route from the originating node to the intended destination. The route request contains a unique identifier, a "source route" through which the route request has progressed, and an intended destination. The source route initially contains the originating node only. Every node that receives the broadcast route request examines the contained identifier and originating node. If the identifier and originating node were recently processed in a single

route request, no further action is taken. Similarly, if the receiving node is already represented in the source route, no action is taken. Otherwise, the receiving node checks to determine if it is the intended destination node. If not, a new route request is rebroadcast with the receiving node appended to the source route. If the receiving node is the final destination, a "route reply" is generated.

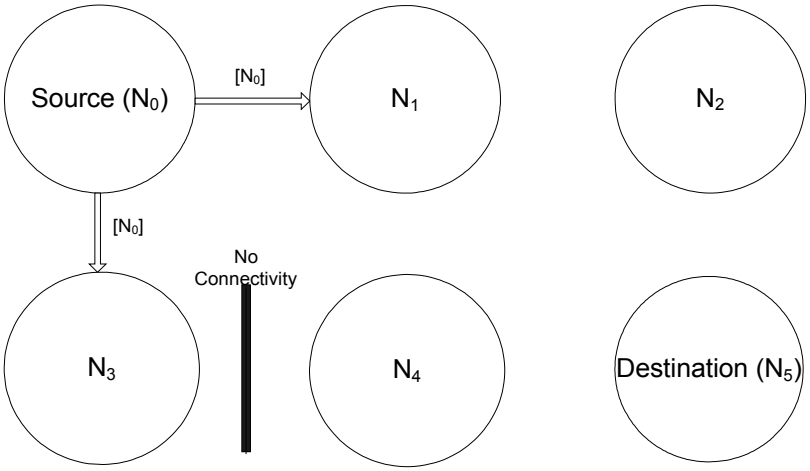

Figure 3: DSR Path Discovery (Step 1)

Figure 3 illustrates a six node network with partial connectivity undergoing the first step of path discovery. The directional arrows between nodes represent route requests. The captions accompanying the arrows represent the partial forward path included within that route request. The diagram indicates the route request has N_0 as the only node in its partial forward path.

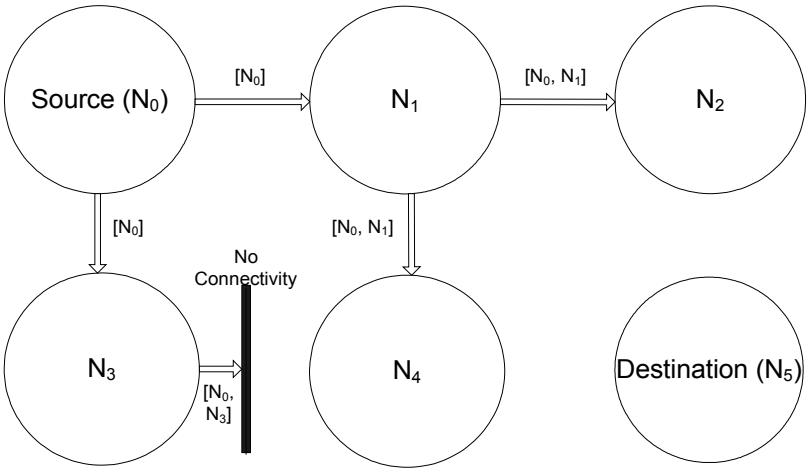

Figure 4: DSR Path Discovery (Step 2)

Figures 4 and 5 illustrate the second and third round of path discovery. Once N_5 receives a route request, it will transmit a route reply and ignore subsequent route requests with the same unique identifier.

Reverse path discovery is used to find a "destination route", which is a path from the intended destination back to the originating node. The destination route is obtained using a process similar to path discovery. The primary distinction between the two is that route replies contain the complete source route discovered during route request proliferation. Nodes receiving the route reply append themselves onto the destination node rather than the source node. By the time the originating node receives a route reply corresponding to its route request, the route reply contains the source route and destination route.

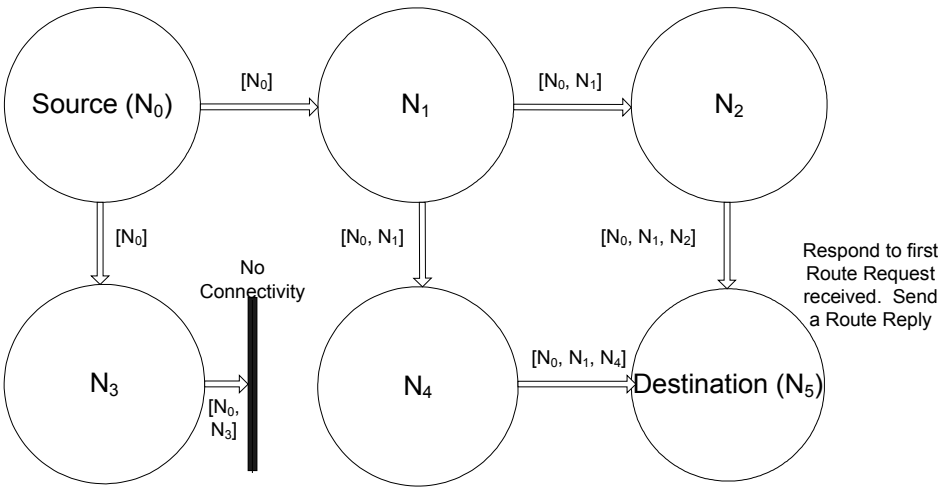

Figure 5: DSR Path Discovery (Step 3)

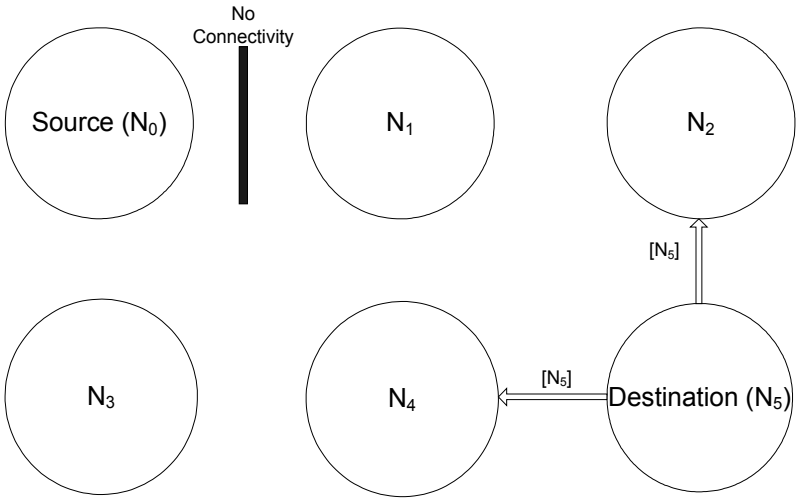

Figure 6: Reverse Path Discovery (Step 1)

When forwarding a route reply R, an intermediate node I may derive additional routes from the available information. If I exists in both the source and destination routes, then I may store a source route to every subsequent node in R's source route. Similarly, I may store a destination route from every node precedent to I in R's destination route. Based on Figure 6, N_4 and N_2 can store a route to and from N_5 conditionally based on whether the forward path embedded within the route reply includes N_4 or N_2.

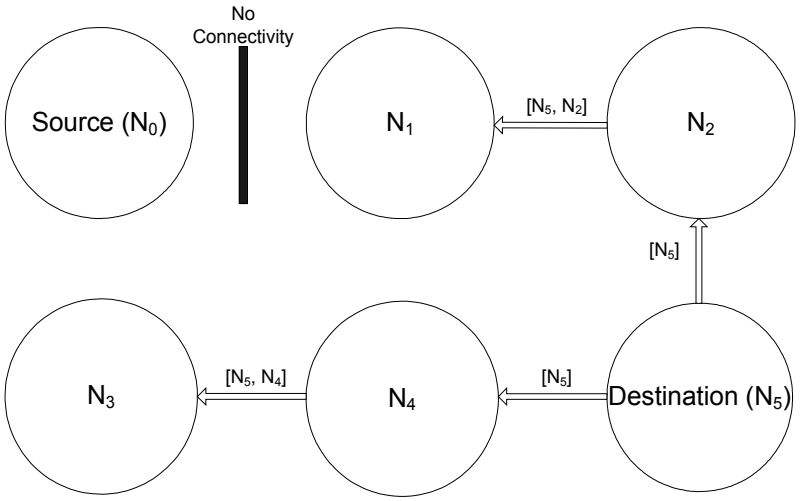

Figure 7: Reverse Path Discovery (Step 2)

Based on Figure 7, N_1 can store a route to and from N_5 when it receives a route reply provided that the forward path within the route reply contains N_1. However, N_3 cannot store a route to and from N_5 under any circumstances because of its limited connectivity during path discovery.

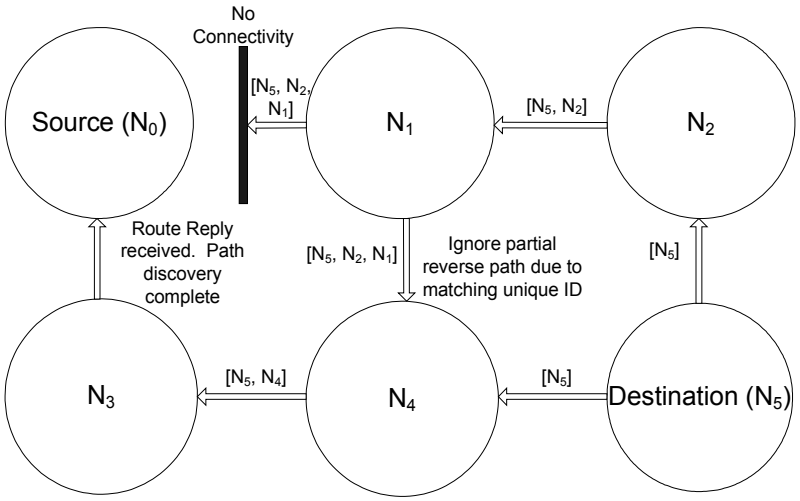

Figure 8: Reverse Path Discovery (Step 3)

Once nodes discover or derive routes from available information, the routes are stored in local caches. In this manner, user data may immediately transmit along stored routes instead of invoking the route discovery process. Routes remain in the cache until they are determined to be broken or until some other unspecified criterion occurs.

The route discovery process over a large, well connected network will cause a large quantity of transmissions to be emitted. DSR contains certain mechanisms to mitigate this. An arbitrary node X may proxy received route requests to destination D if X contains a route to D. In this case, the route from node X to node D is appended to the partial route contained in the route request received by X. Conversely, the route from node D to node X (stored at X) is transmitted from node X in a route reply. This mechanism will limit route request rebroadcasts to the degree that route request receivers contain routes to node D.

The ability for arbitrary nodes to proxy route requests for the intended destination can cause a flood of route reply messages. DSR specifies a random delay to be invoked whenever route requests are proxied. The delay is executed before each node's route reply is transmitted. If a route reply containing a shorter route to the destination node than the monitoring node's route is received during timer execution, the monitoring

node will not send a route reply.

Another mechanism described in DSR is a hop limit. The hop limit prevents route requests and route replies from proliferating too far into the network. Such a mechanism would be useful in networks where a maximum hop count exists between sources and destinations.

Once a source route is obtained to a particular destination, user-submitted data can be forwarded along the route. The destination route back to the source may be included along with the user data. In this manner, application level acknowledgments may be returned from the destination to the source node without initiating the route discovery process.

Route maintenance procedures will be invoked if a node fails to forward user data onto the next node in the route. The basic route maintenance feature in DSR involves using data link protocol mechanisms to determine if the link from node N_i to N_{i+1} is broken. Explicit DSR acknowledgments may be used if no appropriate data link protocol mechanisms exist. If a route is broken, a "route error" notification must be delivered to all nodes in the route prior to N_i. Once received, the route error messages cause the broken route to be removed from the receiver's cache.

Another route maintenance feature is "packet salvaging". Packet salvaging is an additional step that an intermediate node N_i may execute after sending route error notifications. The local route cache at N_i is searched for a replacement route to the same destination as the broken route. If one exists, then the broken portion of the broken route is replaced by the replacement route from N_i to the final destination. At this point, the user data containing the newly reconstructed route is forwarded normally.

DSR would scale well over geographic distances when operating over a line-of-sight network. The hop limit would localize bandwidth consumption due to route discovery from any arbitrary node. In contrast, network consumption would increase proportionally to increases in network density. The hop limit would have no affect on network density increases.

DSR should be configured with a low hop limit for operation over an HF network. This takes advantage of the potential distance covered by node-to-node connections within data link protocols. Additionally, low hop limits limit the length of discovered paths. This is advantageous considering that the likelihood of successfully traversing a multihop route over an HF network decreases in proportion to the length of the route. Factors causing this limitation include interference, channel acquisition delays, and low bandwidth throughout the network. Multiple nodes transmitting simultaneously frequently cause neither transmitted signal to be received. Channel acquisition delays are exacerbated by outgoing data queues containing data intended for transmission to different nodes. Low bandwidth causes round trip time across the network to be proportional to the quantity and length of network traffic.

DSR would scale well over an HF network provided that performance expectations are kept in line with HF channel characteristics. The network will perform optimally when steady state operation (no traffic) is punctuated by occasional route discovery and user data transmission. The network would not scale well to congestion.

2 HF-DSR Overview

HF-DSR is an implementation of DSR that is customized to operate over HF radio channels. It is designed to transfer information to nodes that, due to limitations of HF propagation, cannot receive data directly from a data source. This can occur when two nodes are distant enough that line-of-sight communication is impossible but too close to reflect signals to one another off the ionosphere. In this case, at least one additional HF hop is necessary. More hops may be required if one intermediate node is insufficient.

HF-DSR could conceivably be used to extend the limits of HF propagation range. HF propagation range limits are not typically encountered within existing networks. Instead, networks are limited to well within propagation distance. However, low-power or geographically isolated nodes can use HF-DSR to reach the rest of the network. Additionally, networks may be designed to take advantage of HF-DSR's routing capability.

In order to keep over-the-air control traffic to a minimum, HF-DSR relies on level 2 data link protocols for delivery confirmation services. In most cases, this is sufficient to determine whether or not data has reached subsequent nodes on a route. Due to their scope, level 2 protocols are not capable of autonomously transferring delivery confirmation information across multiple wireless gaps. With this in mind, active route maintenance features such as route error messages and packet salvaging are not implemented within HF-DSR. However, final destinations shall generate route acknowledgments back to the original source. In this manner, the original source can have some level of delivery confirmation.

HF-DSR is designed to maximize the quantity of information derived from the smallest quantity of information received. Similarly to DSR, any HF-DSR node may initiate a route discovery procedure to obtain a route with a specified remote node. Route discovery often involves packet forwarding through one or more intermediate nodes. During this procedure, all intermediate nodes attempt to incorporate received partial routes into their route tables. In this manner, explicit route discovery operations are not always required to populate a node's stored routes.

HF-DSR transmits files rather than packets. This reduces penalties associated with channel acquisition. However, this prevents variable length data payloads from being sent. Variable length payloads are most often associated with real-time streaming. Real-time streaming usually requires relatively constant bandwidth between data sources and destinations. Unfortunately, HF radio networks cannot reasonably guarantee these conditions. As a result, the file-based approach is acceptable.

HF-DSR and DSR use different packet structures and formats. DSR packets are designed with extensibility in mind and are proportionally complex. Optional packet headers allow for variable degrees of functionality. For example, an “acknowledgment request” optional header causes nodes to use DSR-level acknowledgments that simulate level 2 acknowledgments. In contrast, the limited scope of HF-DSR allows for a small number of single purpose packets. With the exception of user transmitted data, HF-DSR packets contains type information, path information, a unique identifier, and message segment information.

3 HF-DSR High-Level Design

3.1.1.1Addressing

HF-DSR relies on level 2 protocol addressing. As a result, there is no need to translate HF-DSR addresses to level 2 protocol addresses and vice-versa.

HF-DSR relies on STANAG 5066 as its data link layer and uses S_PRIMITIVEs to communicate with STANAG 5066. HF-DSR has been tested with the STANAG 5066 implementation within the Harris RF Communications RF-6750W Wireless Gateway (WG). In addition to STANAG 5066, the WG implements HF modem control and radio control This allows MIL-STD-188-141A automatic link establishment to be automatically invoked during node-to-node communications. Additionally, data rates and interleaver settings with the MIL-STD-188-110B modem waveform are automatically adjusted to suit available channel conditions.

It could potentially become necessary to adapt an HF-DSR address scheme distinct from the level 2 protocol address scheme being used. One possibility involves using TCP/IP addressing. In this case, HF-DSR addresses may be converted to STANAG 5066 addresses by using a table lookup mechanism, ARP based mechanism, or some other mechanism..

3.1.1.2Route Discovery

HF-DSR route discovery operates by sending a “path request” packet. The path request contains the following information:

- Packet type information. This field is always set to 0.
- Version information. Currently, this field is set to 1.

- A 32-bit identifier. The path request originator sets the identifier to the value of an automatically incrementing counter that was initialized to a random value.
- A destination node address. The destination node address is set to the node for which a route is to be discovered
- A partial path. The partial path is initialized to contain the path request originator's address.

Once the path request packet is initialized, it is sent to STANAG 5066 to be broadcast to all nodes within range.

Each node N that receives a path request examines it to determine the next course of action. If N has recently processed a path request with the same original source and the same identifier, then no action is taken. Otherwise, if N's address does not match the path requests "destination node address" field, then N's address is appended to the partial path and the path request is re-broadcast. If N matches the destination node address, then a "reverse path request" packet is initialized and submitted to STANAG 5066 for broadcast.

The "reverse path request" packet contains the following information:

- Type information. This field is always set to 1.
- Version information. Currently, this field is set to 1.
- A 32-bit identifier. This gets set to the same value as the identifier in the received path request.
- The path request originator's address.
- The number of addresses in the forward path.
- The forward path. This is set to the "partial path" field within the received path request.
- A partial reverse path. This is initialized to contain N's address.

Once all fields are initialized within the reverse path request, it is submitted to STANAG 5066 for broadcast.

Upon receiving a reverse path request, each node O examines it in a similar manner to how path requests are examined. If O recently processed a reverse path request with the same identifier as the received path request, then no action is taken. Otherwise, if the reverse path request's "original path request originator address" field does not match O's address, then the reverse path request is submitted to STANAG 5066 for re-broadcast. If the reverse path request's "original path request originator address" field matches O's address, then the path discovery procedure is complete. At this point, the forward path and reverse path are both contained within the received reverse path request.

Additional processing takes place when nodes receive reverse path requests. Each node P that receives a reverse path request attempts to derive forward and reverse paths to all nodes in the forward path after P. For each node Q subsequent to P in the forward path, the partial reverse path is searched for an occurrence of Q. If a match is found, then a new route is added to P's address table. The forward path contains all the node in the reverse path request's forward path from P to Q. The reverse path contains all the nodes in the partial reverse path until Q.

Route discovery is required to complete within a certain time period. This s accomplished by a timer that the path request initiator starts immediately after submitting the path request to STANAG 5066. When this "route discovery timer" expires, the route discovery attempt is deemed to be a failure. If a reverse path request is received that causes the route discovery attempt to succeed, the route discovery timer is stopped.

The actual path discovery timer is set by using a constant compiled value and is not configurable from the user interface. The initial timer value is set to 10 minutes, but can be changed to what is required to traverse the radio network.

The route discovery process is expensive in terms of network bandwidth. In order to limit the quantity of bandwidth consumed by route discovery, nodes cannot generate a new path request until all outstanding route discovery attempts have completed. In this manner, an N node network has a maximum of N route discovery attempts active at once.

3.1.1.3Route maintenance

Route maintenance within HF-DSR involves no explicit traffic exchange between nodes. Once discovered, routes are assigned an expiration time. Routes are assumed valid until the expiration time passes. Expired routes may no longer be used to forward data to a particular destination. Instead, a new route discovery procedure must be initiated to find a path that more closely corresponds to existing network conditions.

The actual route expiration timer value is set by using a constant compiled value and is not configurable from the user interface. The initial timer value is set to 10 minutes. Decreasing the value of this constant causes routes to be re-generated more often, increasing network adaptability at the cost of additional path discovery overhead The length of time that newly-discovered routes exist before expiring should be long enough that new routes do not have to be discovered every time a node wishes to transmit data. This period should also be short enough so that channel conditions are reasonably likely to remain consistent throughout its lifetime.

3.1.1.4File Transfer

Data transfer within HF-DSR is limited to file transfer. File transfer is accomplished by breaking the file into multiple segments, each of which is delivered by an HF-DSR data packet. HF-DSR data packets contain the following information:

- Version information. Currently, this field is set to 1.
- Type information. This field is always set to 2.
- The number of addresses in the forward path.
- The forward path.
- The number of addresses in the reverse path.
- The reverse path.
- A 32-bit identifier. The file originator sets the identifier to the value of an automatically incrementing counter that was initialized to a random value.
- The total message size.
- The segment offset.
- The data segment.

The data segment size is derived from the total size of the data packet and the size of the other fields.

Prior to segmentation, files being transmitted are prepended by their null-terminated file name. In this manner, the file name may be extracted my the receiver.

All data packets associated with a file F are transmitted from the source node N_s to the subsequent node N_{s+1} in the forward path. N_{s+1} does not forward any data packets associated with F to N_{s+2} until F is completely received by N_{s+1}. In this manner, N_{s+1} is not required to acknowledge data packets from N_s and forward data packets to N_{s+2} simultaneously. This limits channel access contention among nodes along the forward path.

The 32-bit identifier field is used to associate incoming data packets with one another. In this manner, data packets associated with different files may be simultaneously received and processed correctly.

Once all packets associated with file F are completely received by node N_d, an HF-DSR acknowledgment packet is generated and sent along the reverse path. The acknowledgment packet contains the following information:

- Version information. Currently, this field is set to 1.
- Type information. This field is always set to 3.
- The number of addresses in the reverse path.
- The reverse path. This field is copied from received data packets associated with the file being acknowledged.
- A 32-bit identifier. This value is taken from received data packets associated with the file being

acknowledged.

The acknowledgment packet is transmitted from node N_d to node N_s along the reverse path. A node N_i along both the forward and reverse paths may use use the 32-bit identifier to verify paths from N_i to N_d and from N_d to N_i. Node N_s responds to a received acknowledgment packet by verifying the identifier with that of the transferred file F. If the identifiers match, then N_s considers its file transfer to be complete.

3.1.1.5Packet Design

Note that STANAG 5066 provides the size of incoming HF-DSR packets. The size of constant-length fields is known The difference between incoming packet size and constant size yields a quantity useful to determine the size of at least one variable-length field.

3.1.1.5.1.1Path Request Packet

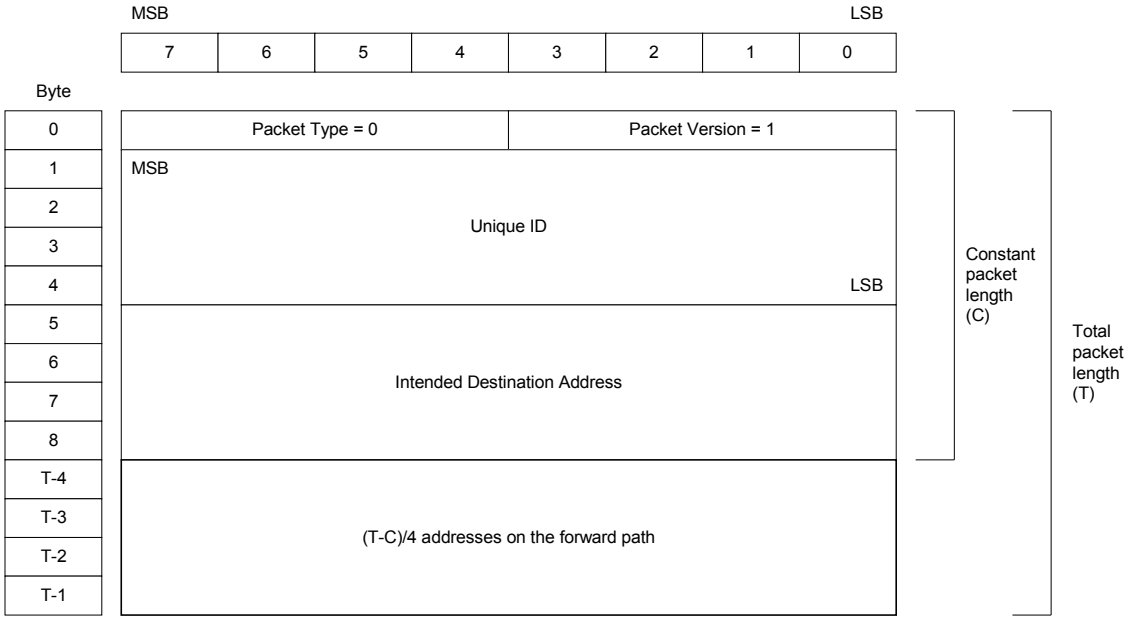

Figure 9: HF-DSR path request with two addresses on the forward path

3.1.1.5.1.2Reverse Path Request Packet

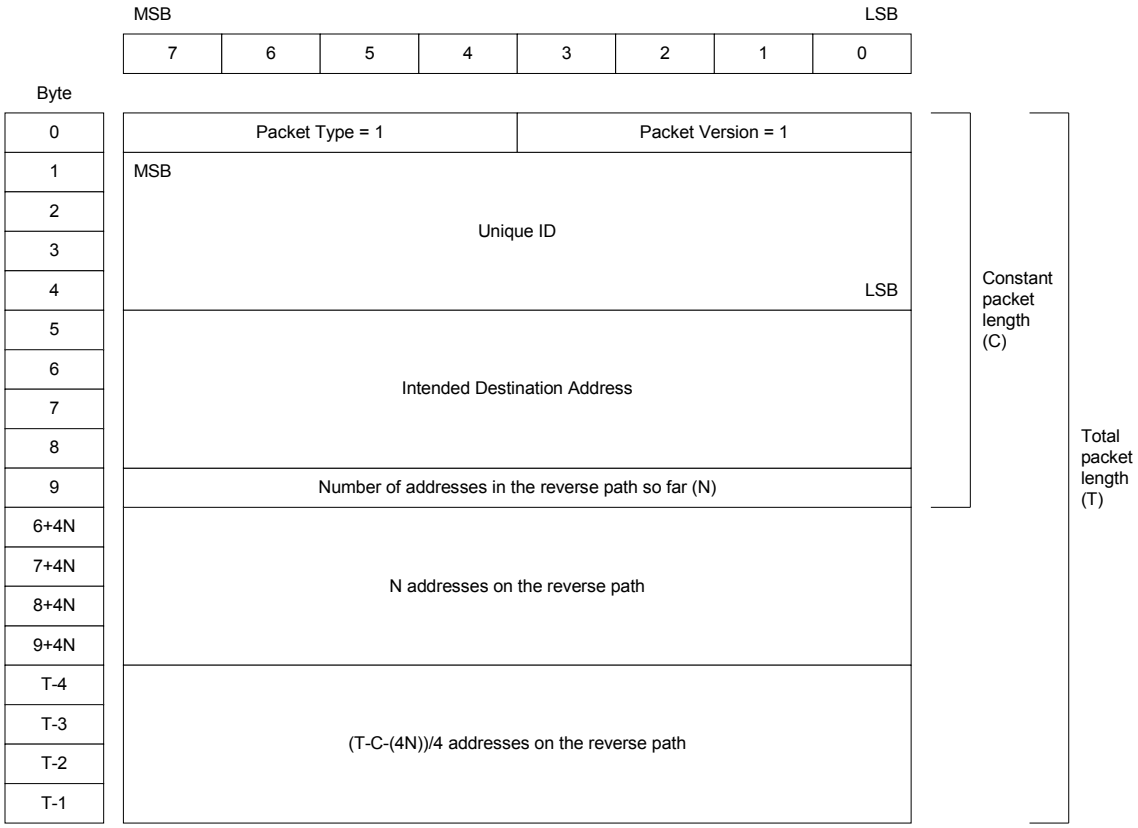

Figure 10: HF-DSR reverse path request

3.1.1.5.1.3Data Packet

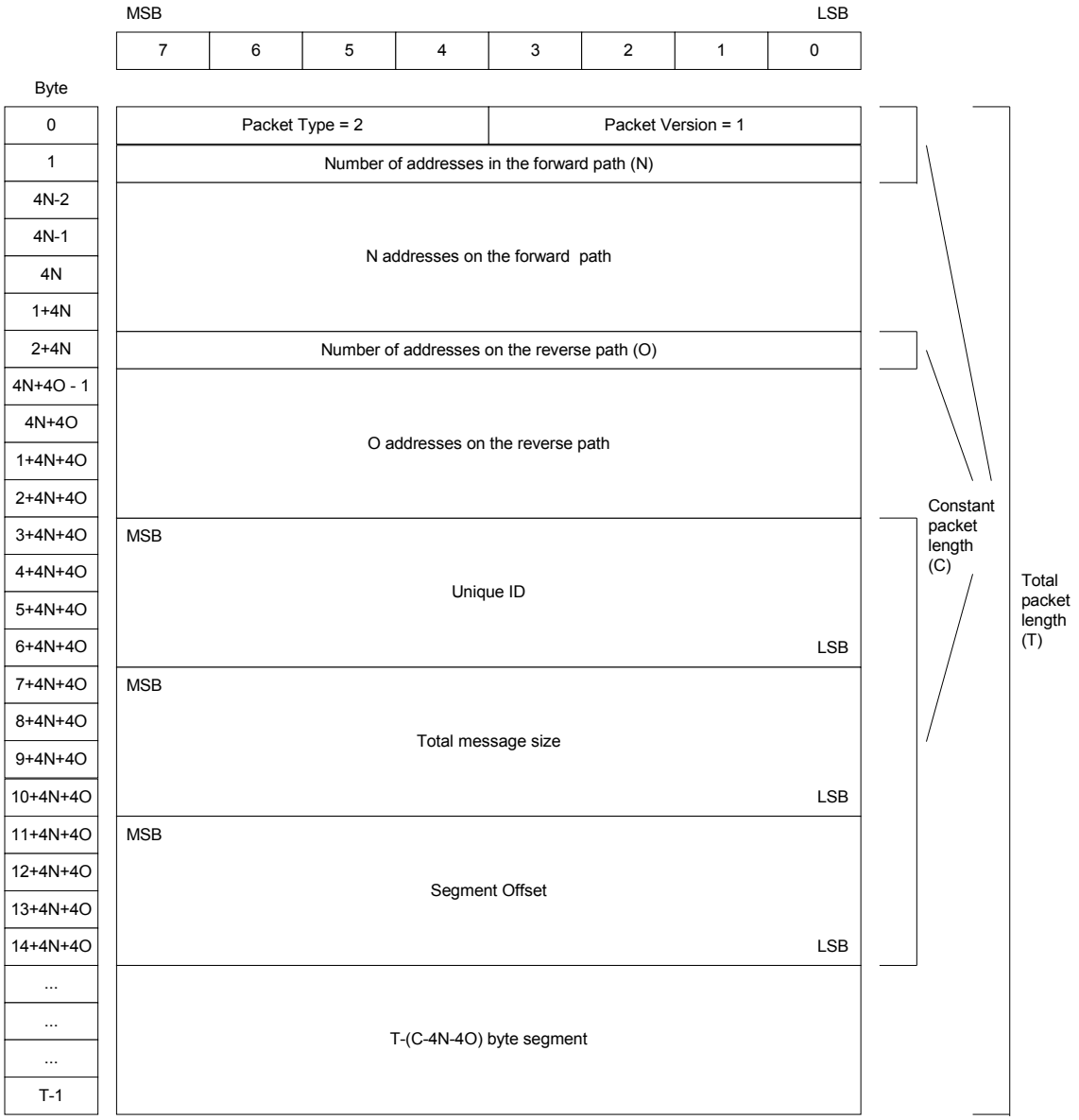

Figure 11: HF-DSR data packet

3.1.1.5.1.4Acknowledgment packets

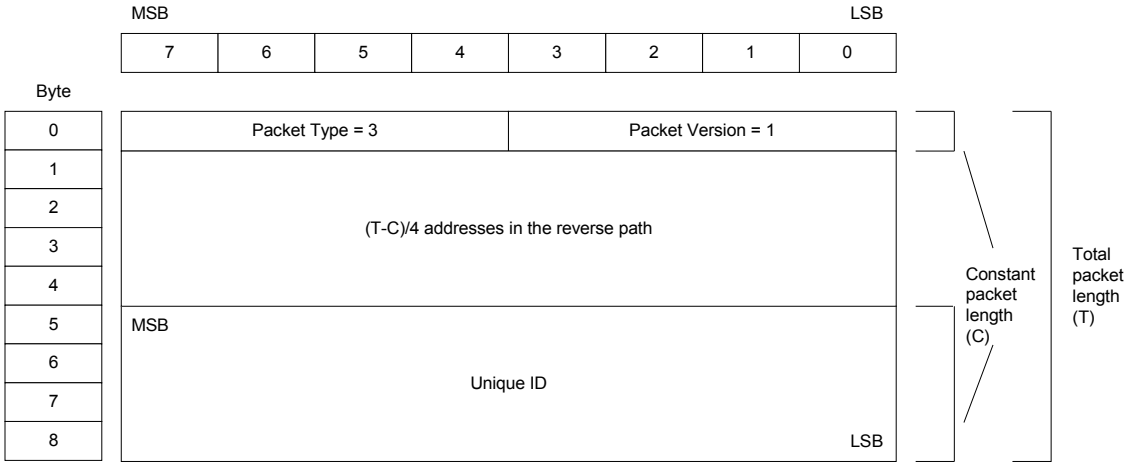

Figure 12: HF-DSR acknowledgment packet

3.1.1.6STANAG 5066 Interface

HF-DSR packets are transmitted and received through a STANAG 5066 interface. The STANAG 5066 interface consists of a TCP/IP connection with port 5066 on the host containing the STANAG 5066 server. S_PRIMITIVEs are exchanged between HF-DSR and STANAG 5066 to negotiate an active connection on a "subnet access point" (SAP). Once a connection is successfully negotiated, S_PRIMITIVEs are exchanged to transfer data over a radio channel.

There are 16 subnet access points per STANAG 5066 server. Each subnet access point may be bound by zero or one client at once. Clients send an S_BIND_REQUEST S_PRIMITIVE to STANAG 5066 which requests to be bound on a particular subnet access point.

Figures 13 through 19 are taken from the STANAG 5066 specification [STANAG 5066].

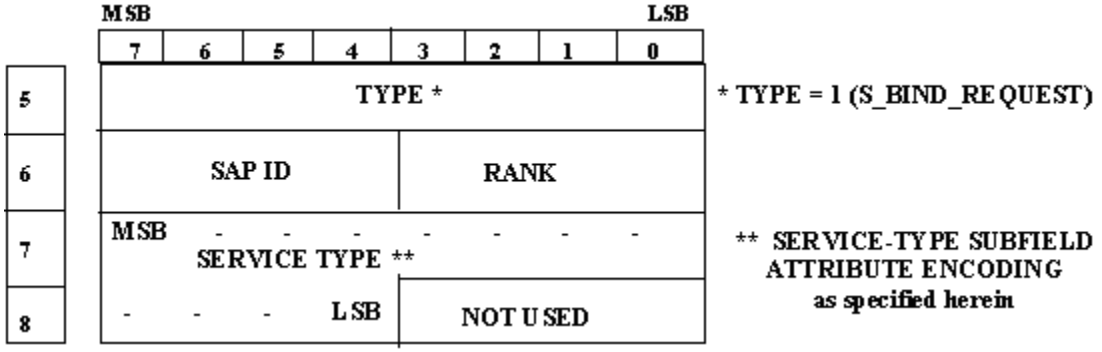

Figure 13: STANAG 5066 Bind Request S_PRIMITIVE

The bind request S_PRIMTIVE also contains a "rank", which is used to determine whether a client already bound on a particular SAP should be unbound in favor of the newly binding client. The highest rank is 15 and the lowest rank is 0. HF-DSR currently binds with a rank of 1.

The bind request also contains a "service type" field. The service type is a complex field that includes default data delivery mode and other related options These options include whether data should be delivered ARQ or non-ARQ, the number of non-ARQ repeats, and whether delivery confirmation is enabled.

STANAG 5066 responds to a "bind request" S_PRIMITIVE with either a "bind accepted" or "bind rejected" S_PRIMITIVE. The "bind accepted" S_PRIMTIVE contains the SAP ID that a client bound on

and a MTU, or maximum transmittable unit. The MTU is typically 2048 kilobytes for mixed ARQ and non-ARQ traffic.

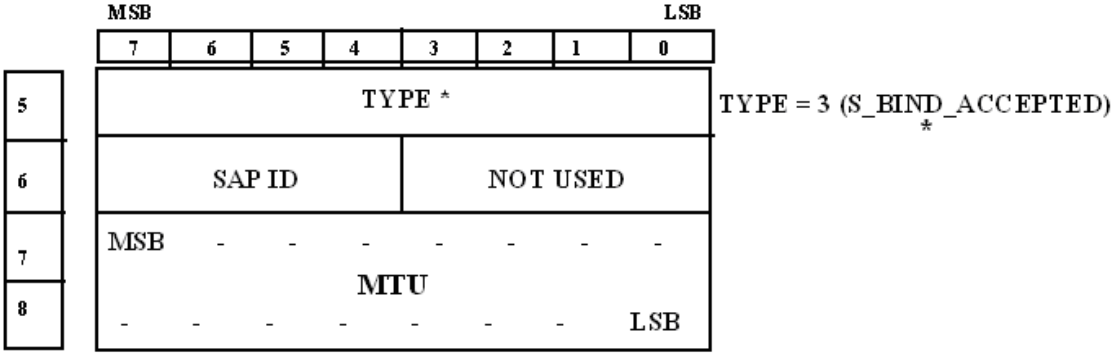

Figure 14: STANAG 5066 Bind Accepted S_PRIMITIVE

The "bind rejected" S_PRIMITIVE contains a "reason" field. This field enumerates the reasons why a bind attempt failed.

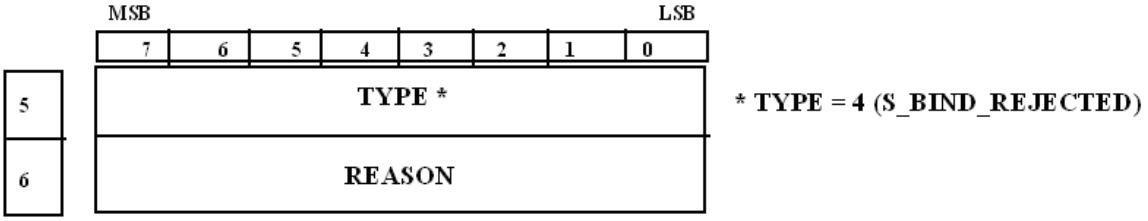

Figure 15: STANAG 5066 Bind Rejected S_PRIMTIVE

The "unbind request" S_PRIMITIVE is sent by a client when it unbinds. No S_PRIMITIVE response is made by STANAG 5066 upon receiving an unbind request. Instead, STANAG 5066 unregisters the client from its subnet access point.

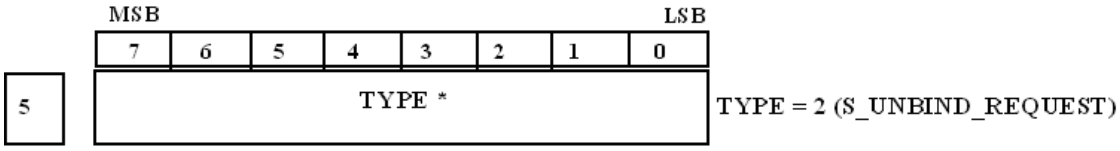

Figure 16: STANAG 5066 Unbind Request S_PRIMITIVE

The "unbind indication" S_PRIMITIVE is sent by STANAG 5066 when a client is to be forcibly unbound. This typically occurs when a client binds to a SAP that is already bound. If the new client is to replace the previously bound client, STANAG 5066 sends an "unbind indication" S_PRIMITIVE to the previously bound client.

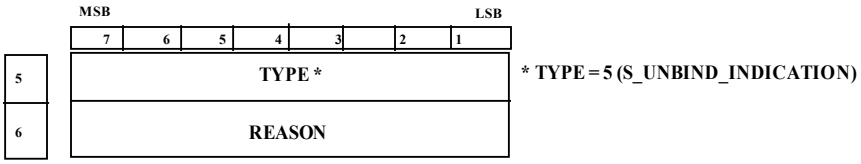

Figure 17: STANAG 5066 Unbind Indication S_PRIMITIVE

Once HF-DSR is bound to STANAG 5066, it may submit data to be transferred over radio channels. Data is submitted to STANAG 5066 by using the "unidata request" S_PRIMITIVE.

The unidata request S_PRIMITIVE contains a "priority" field. This allows differently submitted data to differentiate itself by precedence. HF-DSR consistently sets the priority field to 0.

The “unidata request” S_PRIMITIVE also contains a “Destination SAP Id” field. This field specifies the SAP to deliver the data to upon receipt. HF-DSR sets this option to be the same SAP at which HF-DSR is bound to STANAG 5066.

A “Destination Node Address” field within the unidata request indicates the destination node or nodes. HF-DSR sets this field to an individual node address in the case of directed data transfer along previously discovered paths. In the case of route discovery, HF-DSR sets the field to a user-configured group address.

The “Delivery Mode” field allows for custom delivery parameters per data payload. This allows individual data submissions to vary in data confirmation mode and in-order delivery requirements. HF-DSR sets the “Delivery Mode” field to ARQ, in-order packet delivery, and node delivery confirmation when files are being transferred along previously discovered paths. HF-DSR sets the “Delivery Mode” field to non-ARQ, as-they-arrive packet delivery, and no delivery confirmation when routes are being discovered.

The “Time To Live” field allows S_PRIMITIVEs to expire if they have not been successfully transmitted before a certain time has elapsed. HF-DSR sets this field to never expire. The “Size of U_PDU” field represents the size of the U_PDU in bytes. Finally, the U_PDU field contains the actual data segment to be delivered.

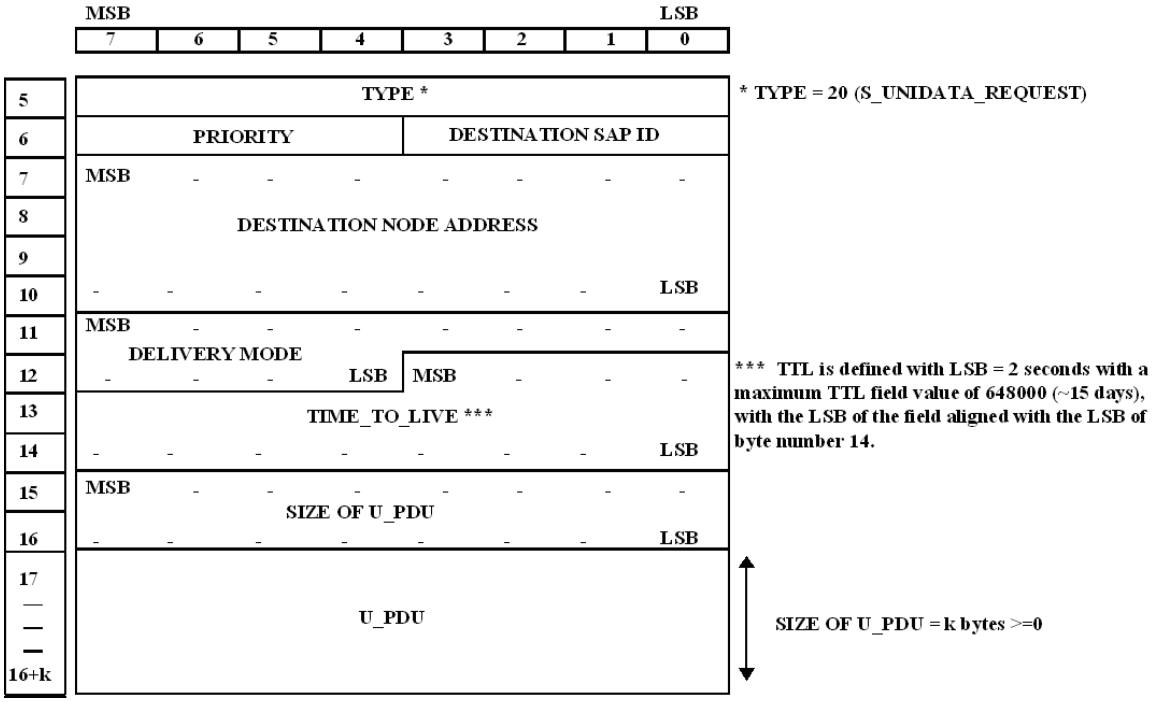

Figure 18: STANAG 5066 Unidata Request S_PRIMITIVE

Incoming HF-DSR transfers are delivered from STANAG 5066 to peer HF-DSR nodes by using the “unidata indication” S_PRIMITIVE. The unidata indication contains information about the data being received as well as the data itself.

The “destination node” field indicates whether the “unidata indication” S_PRIMI TIVE is intended for a single node or multiple nodes. As such, the “destination node” field contains either an individual address, which is used during file transfer, or a group address, which is used during route discovery.

The “transmission mode” field within the “unidata indication” S_PRIMITIVE indicates whether the S_PRIMITIVE represents an ARQ or non-ARQ reception. “Unidata indication” S_PRIMITIVEs associated with HF-DSR route discovery have their “transmission mode” field set to non-ARQ. Conversely, “unidata indication” S_PRIMITIVEs associated with HF-DSR file transfer have their “transmission mode” field set to ARQ.

“Unidata indication” S_PRIMITIVEs contain a “source SAP id” field. Since HF-DSR requires each node to bind to the same SAP, this field shall be consistent for all types of traffic.

“Unidata indication” S_PRIMITIVEs contain a “source node address” field. This field is universally set to the node address that sent the corresponding “unidata request” S_PRIMITIVE.

The “unidata indication” S_PRIMITIVE contains a “size of U_PDU” field. STANAG 5066 shall set this field to the number of bytes contained in the corresponding U_PDU field within the S_PRIMITIVE.

The “unidata indication” S_PRIMITIVE contains “number of errored blocks” and “number of non-received blocks” fields. STANAG 5066 sets these fields to 0, indicating that the “pointers and sizes of errored blocks” and “pointers and sizes of non-received blocks” fields are non-existent. These fields are designed to indicate when a U_PDU has been partially received. HF-DSR does not request a transmission mode that could allows U_PDUs to be received with errors.

The “U_PDU” field within the “unidata indication” S_PRIMTIVE contains the data received from HF-DSR. The receiving HF-DSR implementation is responsible for interpreting the U_PDU appropriately.

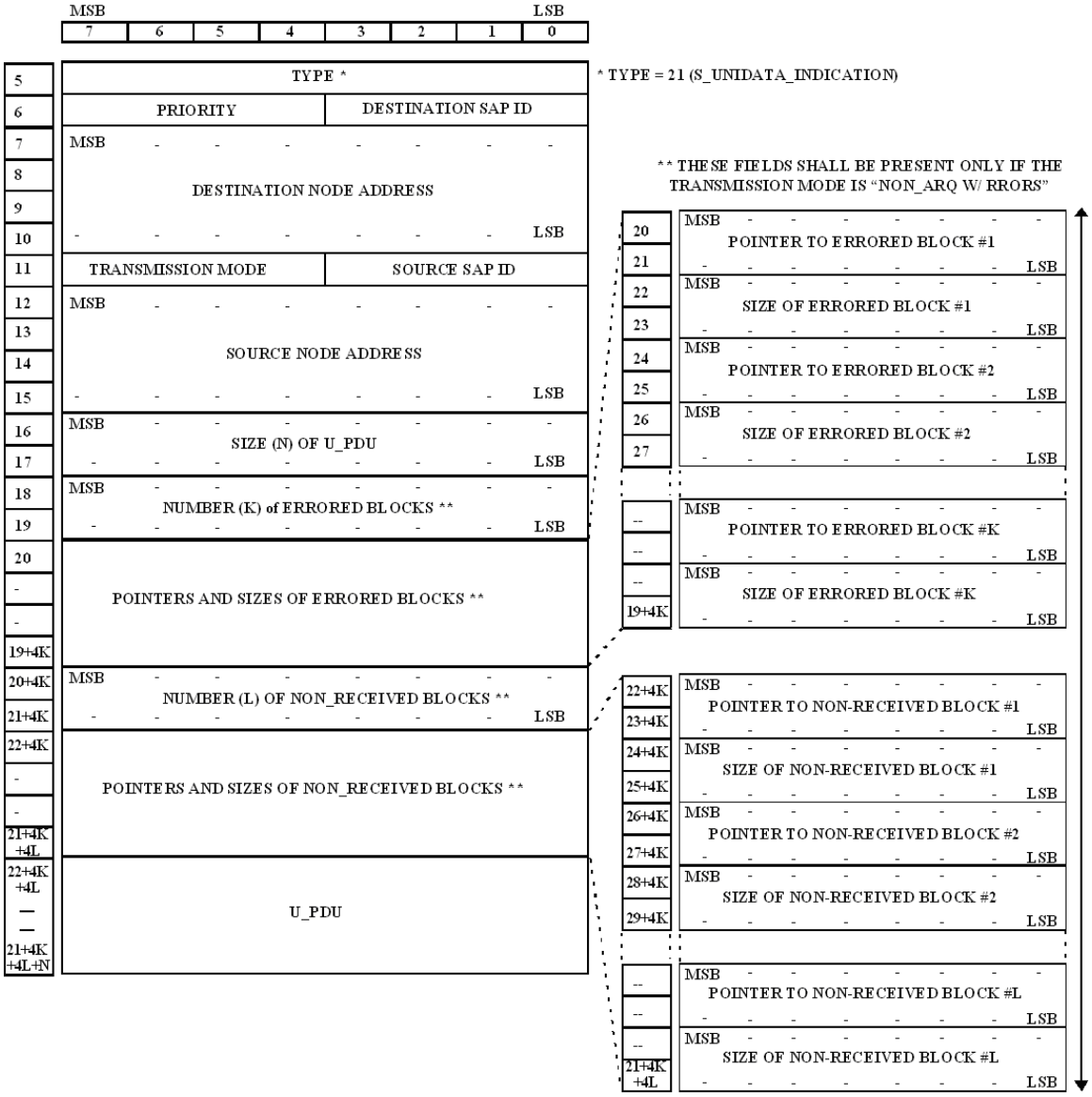

Figure 19: STANAG 5066 Unidata Indication S_PRIMITIVE

4 HF-DSR Software Design

4.1.1.1Class Overview

HF-DSR has been implemented in C# for the purposes of this project. The following classes have been created to accommodate this:

- formHFDSR – This class inherits from the "Form" class and provides user interface functionality.
- hf_dsrSession – This class acts as a translation layer between the user interface and the rest of HF-DSR. In this manner, an abstraction layer separates the user interface and the protocol classes.
- hf_dsrProtocol – This class contains most of the protocol logic required to establish routes. It also contains some of the logic required to send and receive messages.
- hf_dsrDataLinkInterface – This class contains functionality to issue S_PRIMITIVEs to the socket wrapper class. Additionally, incoming data from the socket wrapper can be identified and sent to the protocol. Additionally, hf_dsrDataLinkInterface contains most of the logic that binds and unbinds from STANAG 5066.

- ThreadedSocketWrapper – This class wraps the TCP/IP socket connecting HF-DSR to STANAG 5066. Incoming data from STANAG 5066 is handled on worker threads. A separate thread queues and sends outgoing data to STANAG 5066. Note that this class is generically written and is used by other software components.
- TimestampTraceListener – This class registers with the .NET "Trace" class and provides functionality to log events to a file along with a timestamp indicating when the events were logged. Clients can log to the static Trace class to execute TimestampTraceListener.
- hf_dsrPacket – This class provides common packet serialization and de-serialization functionality. It is the superclass of all other HF-DSR packet classes.
- hf_dsrAck – This class can serialize and de-serialize HF-DSR acknowledgment packets.
- hf_dsrData – This class can serialize and de-serialize HF-DSR data packets.
- hf_dsrPathRequest – This class can serialize and de-serialize HF-DSR path request packets.
- hf_dsrReversePathRequest – This class can serialize and de-serialize HF-DSR reverse path requests.
- hf_dsrMessage – This class handles message creation, transmission, reception, and re-transmission.
- S5066S_Primitive – This class provides common base class functionality for all S_PRIMITIVE classes.
- S5066Address – This class serializes, de-serializes, and validates STANAG 5066 addresses.
- S5066BindAccept – This class serializes and de-serializes STANAG 5066 S_BIND_ACCEPT S_PRIMITIVEs.
- S5066BindRequest – This class serializes and de-serializes STANAG 5066 S_BIND_REQUEST S_PRIMITIVEs.
- S5066UnbindRequest – This class serializes and de-serializes STANAG 5066 S_UNBIND_REQUEST S_PRIMITIVEs.
- S5066UnidataIndication – This class serializes and de-serializes STANAG 5066 S_UNIDATA_INDICATION S_PRIMITIVEs.
- S5066UnidataRequest – This class serializes and de-serializes STANAG 5066 S_UNIDATA REQUEST S_PRIMITIVEs.

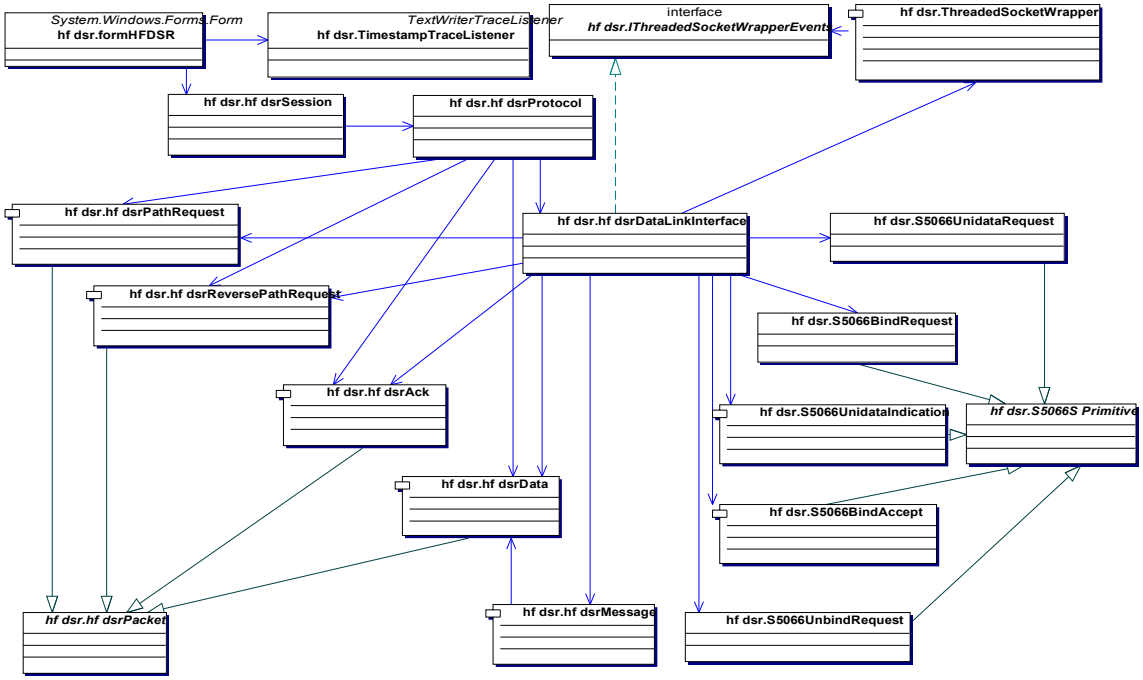

Figure 20: HF-DSR Class Relationships

4.1.1.1.1.1formHFDSR Class

A single instance of the formHFDSR class is responsible for displaying the main application window and interfacing with the hf_dsrSession object. User interface events cause notifications to hf_dsrSession and application events cause hf_dsrSession to notify formHFDSR. In this manner, there is a degree of

abstraction between the user interface and functional portions of the application.

The formHFDSR class contains the following methods:

- Constructor – This method initializes the log file and the hf_dsrSession object.
- Dispose – This method uninitializes logging and signals the hf_dsrSession.
- Main – This is the HF-DSR entry point. It creates a formHFDSR object and passes it to the static Application class for execution.
- formHFDSR_Load – This method is called when the form finishes loading. It initializes all the text fields.
- buttonBind_Click – This method is called when the "bind" button is clicked. The appropriate values are read in from text boxes. Next, the bind request is passed to the hf_dsrSession object.
- SetStatusUnbound – This method is called by the hf_dsrSession object to indicate that the application has entered the "unbound" state. The user-interface is set appropriately.
- SetStatusBound – This method is called by the hf_dsrSession object to indicate that the application has entered the "bound" state. The user-interface is set appropriately.
- SetStatusMsgComplete – This method is overloaded to represent outgoing and incoming message complete notifications. In either case, the user-interface is set appropriately.
- SetStatus – This method allows the hf_dsrSession object to set a text string into the user interface.
- OnRouteListUpdate – This method is called by the hf_dsrSession object to indicate that the list of routes has been updated.
- buttonSelectFile_Click – This method is called when the "Select File" button is clicked. First, a file dialog is created. If the user selects a file, then the filename is set into the user interface.
- buttonSend_Click – This method is called when the "Send File" button is clicked. First, the filename and remote address are read from the user interface. Next, a send request is directed toward the hf_dsrSession object.
- buttonDiscoverPath_Click – This method is called when the "Discover Path" button is clicked. First, the remote address is read from the user interface. Next, a discover path request is directed toward the hf_dsrSession object.

4.1.1.1.1.2S5066Address Class

The S5066Address class represents a STANAG 5066 address. These addresses are numerical quantities encompassing 28bits and typically represented in the form "A.B.C.D". In this case, A may range from 0 through 15, and B, C, and D may range between 0 and 255. STANAG 5066 uniquely identifies nodes by STANAG 5066 addresses. Additionally, it is possible for multiple STANAG 5066 interfaces on the same node to be identified by distinct addresses.

The S5066Address class contains functionality to validate addresses, serialize and de-serialize addresses from byte arrays, and convert addresses from string representations. Additionally, S5066Address implements the Object.ToString() method. In this manner, S5066Address objects can be easily displayed in "A.B.C.D" format.

The S5066Address class contains the following methods:

- Constructor – This constructor takes a byte array containing a STANAG 5066 address and an integer representing the address' offset into the array. The constructor fills the object's instance data based on the contents of the array at the designated location.
- Constructor – This constructor takes a string representing a STANAG 5066 address and a boolean indicating whether the address is a group address or an individual address. The constructor verifies the string for correct formatting
- Serialize – This method takes a byte array in which the S5066Address object is to be serialized and an integer representing the offset into the byte array to begin serializing. The S5066Address object is written into the byte array using the format specified in STANAG 5066 Section A.2.2.28.1.
- ToString – This method is overridden from Object.ToString() and converts the S5066Address object into the abovementioned "A.B.C.D" format. The resulting string is returned.

- Equals – This method takes an Object reference representing an object to be tested for equality. Equals() is overridden from Object.Equals() and determines whether the supplied Object reference is a S5066Address with an identical value. If so, Equals() returns "true". Otherwise, Equals() returns "false".
- GetHashCode – This method is overridden from Object.GetHashCode(). GetHashCode() forces two S5066Address objects that cause Equals() to return "true" to yield the same GetHashCode() return value. In this manner, .NET data structures that rely on GetHashCode() to determine equality work correctly with S5066Address.

4.1.1.1.1.3hf_dsrSession Class

The hf_dsrSession class is instantiated once and provides communication between an hf_dsrProtocol object and a formHFDSR object. Frequently, communication between the two classes requires translation. The hf_dsrSession class provides these services.

A state machine is contained within hf_dsrSession that handles inputs differently based on the current state. There are three states: "Unbound", "Bound", and "Initial". Each state is represented by a C# delegate and an associated method. Additionally, the current state is represented by a delegate. In this manner, invoking the instance variable representing the current state causes the appropriate method to be invoked.

Each state machine input requires a different number of parameters to the state machine. These parameters are stored in an array of generic objects that is re-created for each state machine invocation. In this manner, input handlers within the state machine methods extract as many parameters as are needed for each input.

Each state machine method handles a subset of available inputs. Invoking a state machine method with an unhandled input causes a notification to be logged.

The state machine handles the following inputs based on user interface events:

- SMI_USER_START_REQUEST – This input represents a request to initialize the application.
- SMI_USER_BIND_REQUEST – This input represents a bind or an unbind request.
- SMI_USER_SEND_REQUEST – This input represents a file send request.
- SMI_USER_QUIT_REQUEST – This input represents a request to shut down the application.
- SMI_USER_DISCOVER_PATH_REQUEST – This input represents a discover path request.

The state machine handles the following inputs based on protocol events:

- SMI_INCOMING_MESSAGE_COMPLETED – This input represents a received message intended for the local node.
- SMI_OUTGOING_MESSAGE_COMPLETED – This input represents a received acknowledgment packet that corresponds with an outstanding transfer initiated by the local node.
- SMI_ROUTE_DISPLAY_UPDATE – This input represents a change in the routes stored at the local node.
- SMI_SHOW_STATUS – This input represents an arbitrary status report.

The hf_dsrSession class contains the following methods:

- Constructor – The constructor initializes the three delegates associated with the session state machine. Additionally, the initial state is set. Finally, an hf_dsrProtocol object is initializes and provided with a reference to the hf_dsrSession.
- Start – This method is called from formHFDSR and posts an SMI_USER_START_REQUEST input to the state machine. No parameters are assigned to the array of generic objects.
- OnBindRequest – This method is called from formHFDSR and takes a decimal SAP ID, a string containing a STANAG 5066 individual address, and a string containing a STANAG 5066 group address. The string parameters are converted to S5066Address objects and validated. Next, the array of state machine parameters is filled with the SAP ID and S5066Address objects. Finally, an SMI_USER_BIND_REQUEST input is passed to the state machine.

- OnSendRequest – This method is called from formHFDSR and takes a string containing a path to a file, a string containing a remote address, and a boolean indicating whether the transfer should be ARQ or non-ARQ. The remote address is converted from a string to a S5066Address object and validated. Next, the array of state machine parameters is filled with the S5066Address object, the file path, and the boolean. Finally, an SMI_USER_SEND_REQUEST input is passed to the state machine.
- OnDiscoverPathRequest – This method is called from formHFDSR and takes a string containing a remote address to discover. First, the remote address is converted into a S5066Address object and validated. Next, the S5066Address object is placed into the array of state machine parameters. Finally, an SMI_USER_DISCOVER_PATH_REQUEST input is passed to the state machine.
- OnQuitRequest – This method is called from formHFDSR. It passes an SMI_USER_QUIT_REQUEST input to the state machine.
- OnPathUpdateFromProtocol – This method is called from hf_dsrProtocol and is passed two two-dimensional arrays of S5066Addresses. The first array is a vector of forward paths. The second array is a vector of reverse paths. Both arrays are placed into the array of state machine parameters. Next, a SMI_ROUTE_DISPLAY_UPDATE input is passed to the state machine.
- OnMessageReceived – This method is called from hf_dsrProtocol and is passed a stream of message contents, a string containing the filename, and a S5066Address representing the message source. First, these parameters are placed into the array of state machine parameters. Next, a SMI_INCOMING_MESSAGE_COMPLETED input is passed to the state machine.
- OnOutgoingMessageComplete – This method is called from hf_dsrProtocol and is passed a string containing a filename. The string is placed into the array of state machine parameters. Next, an SMI_OUTGOING_MESSAGE_COMPLETED input is passed to the state machine.
- OnShowStatus – This method is called from hf_dsrProtocol and takes a string that contains status. The string is placed into the array of state machine parameters. Next, an SMI_SHOW_STATUS input is placed into the state machine.
- InitialState – This method represents the “Initial” state within the state machine. SMI_USER_START_REQUEST is the only input handled in this state. Processing SMI_USER_START_REQUEST causes causes formHFDSR to display appropriately for “Unbound” state. Next, the current state to be set to “Unbound”.
- UnboundState – This method represents the “Unbound” state within the state machine. Three inputs are accepted in this state:
    - The SMI_USER_BIND_REQUEST handler extracts a decimal SAP ID, a S5066Address object representing the local node's individual address, and the local node's group address from the array of state machine parameters. Next, the handler invokes the Bind method on the hf_dsrProtocol object.
    - The SMI_USER_QUIT_REQUEST handler performs no explicit action.
    - The SMI_SHOW_STATUS handler extracts a status string from the array of state machine parameters. Next, the handler invokes the SetStatus method on formHFDSR with the status.
- BoundState – This method represents the “Bound” state within the state machine. Eight inputs are accepted in this state:
    - The SMI_USER_BIND_REQUEST handler invokes the Unbind method on hf_dsrProtocol. Next, the current state gets changed to “Unbound”.
    - The SMI_USER_QUIT_REQUEST handler invokes the Unbind method on hf_dsrProtocol. Next, the current state gets changed to “Unbound”.
    - The SMI_USER_SEND_REQUEST handler extracts a S5066Address object representing the destination address, a string containing a file path, and an ARQ/non-ARQ boolean from the array of state machine parameters. Next, the SendFile method is invoked on hf_dsrProtocol.
    - The SMI_USER_DISCOVER_PATH_REQUEST handler extracts a S5066Address object from the array of state machine parameters. The S5066Address object represents the node to which a path is to be discovered. Next, the DiscoverPath method is invoked on hf_dsrProtocol.
    - The SMI_INCOMING_MESSAGE_COMPLETED handler extracts a Stream object representing the received file, a string containing its filename, and a S5066Address object representing the source node from the array of state machine parameters. The file is saved to a “HF-DSR Received Files” subdirectory under the user's personal folder. Next, the SetStatusMsgComplete method is invoked on formHFDSR.
    - The SMI_OUTGOING_MESSAGE_COMPLETED handler extracts a string representing the

outgoing file name from the array of state machine parameters. Next, the SetStatusMsgComplete method is invoked on the formHFDSR object.
- The SMI_ROUTE_DISPLAY_UPDATE handler extracts two two-dimensional arrays of S5066Address objects from the array of state machine parameters. The first 2-D array represents forward paths currently stored at the local node. The second 2-D array represents reverse paths currently stored at the local node. These arrays are formatted into a string. Next, the OnRouteListUpdate method is invoked on formHFDSR.
- The SMI_SHOW_STATUS handler extracts a status string from the array of state machine parameters. Next, the handler invokes the SetStatus method on formHFDSR with the status.

4.1.1.1.1.4hf_dsrProtocol Class

The hf_dsrProtocol class coordinates transmission and reception of HF-DSR packets to facilitate route discovery, route maintenance, and message transfer. hf_dsrProtocol stores discovered forward and reverse paths, maintains timers for route discovery and file transfer, and participates in binding and unbinding from STANAG 5066. hf_dsrProtocol reports events to hf_dsrSession, passes hf_dsrData packets to hf_dsrMessage for processing, and forwards bind and unbind requests to hf_dsrDataLinkInterface. Additionally, hf_dsrProtocol receives commands from hf_dsrSession and incoming data from hf_dsrDataLinkInterface.

The hf_dsrProtocol class uses hash tables to store forward and reverse paths by destination node, messages by unique identifier, incoming routes by remote node, and path expiration times by remote node. In this manner, data access operations can take place using as little processing as possible.

A state machine is contained within hf_dsrProtocol that is constructed similarly to the state machine within hf_dsrSession. Four states are represented within the hf_dsrProtocol state machine: “Idle”, “Waiting for Ack”, “Waiting for Route to Send”, and “Waiting for Route”.

The state machine handles the following inputs related to user initiated commands:

- PSI_START_SENDING – This input represents a file send request.
- PSI_DISCOVER_PATH – This input represents a discover path request.

The state machine handles the following inputs related to network events:

- PSI_PATH_REQUEST_RECEIVED – This input represents an incoming path request packet.
- PSI_REVERSE_PATH_REQUEST_RECEIVED – This input represents an incoming reverse path request packet.
- PSI_DATA_RECEIVED – This input represents an incoming data packet.
- PSI_ACK_RECEIVED – This input represents an incoming acknowledgment packet.

The state machine handles the following inputs related to timer events:

- PSI_WAITING_FOR_ACK_TIMER_EXPIRED – This input represents the maximum duration of a file transfer being exceeded.
- PSI_WAITING_FOR_ROUTE_TIMER_EXPIRED – This input represents the maximum duration of route discovery being exceeded.

The hf_dsrProtocol class contains the following methods:

- Constructor – This method takes a reference to a hf_dsrSession object representing an event sink. First, the hf_dsrSession reference is stored as an instance variable. Next, all necessary class variables are initialized. This process includes state machine, timer, and hash table initialization.
- StartWaitingForAckTimer – This method takes an unsigned integer representing a quantity of milliseconds. A “Timer” object representing the “Waiting for Ack” timer is reset and started with the indicated timeout.

- StartWaitingForRouteTimer – This method takes an unsigned integer representing a quantity of milliseconds. A “Timer” object representing the “Waiting for Route” timer is reset and started with the indicated timeout.
- WaitingForAckTimerExpired – This method takes a placeholder “Object” object to conform to the “Timer” class expiration delegate. The state machine is subsequently invoked with the PSI_WAITING_FOR_ACK_TIMER_EXPIRED input.
- WaitingForRouteTimerExpired – This method takes a placeholder “Object” object to conform to the “Timer” class expiration delegate. The state machine is subsequently invoked with the PSI_WAITING_FOR_ROUTE_TIMER_EXPIRED input.
- UpdateSessionWithPaths – This method creates two two-dimensional arrays of S5066Addresses to represent the forward and reverse paths. These arrays are filled by iterating through the hash tables of forward and reverse paths. Next, the OnPathUpdateFromProtocol is invoked on hf_dsrSession.
- GetPathsByDestination – This method takes an array of S5066Address objects representing the forward path, an array of S5066Address objects representing the reverse path, and a S5066Address object representing a final destination. The two arrays of S5066Address objects are out parameters and are passed by reference. If a forward path and reverse path exist for the destination address, then the paths are validated against their expiration times. If the paths have not expired, then the reference parameters are filled in and the method returns “true”. In all other cases, the method returns “false”.
- CalculateAndStoreRoutes – This method takes an array of S5066Address objects representing the forward path and an array of S5066Address objects representing the reverse path. If the local node N is on both the forward and reverse path, then a route is stored from N to O and from O to N for each node O that appears on the forward path subsequently to N and on the reverse path prior to N.
- SendReversePathRequest – This method takes a S5066Address object representing the reverse path request's final destination, an array of S5066Address objects representing a complete forward path, an array of S5066Address objects representing a reverse path accumulated so far, and an unsigned integer representing a unique identifier. An hf_dsrReversePathRequest object is constructed with the supplied parameters, the stored group address, and a reference to the hf_dsrDataLinkInterface class. Next, the “StartSending” method is invoked on the reverse path request.
- SendPathRequest – This method takes a S5066Address object representing the path request packet's final destination, an array of S5066Address objects representing the forward path accumulated so far, and an unsigned integer representing a unique identifier. An hf_dsrPathRequest object is constructed with the supplied parameters, the stored group address, and a reference to the hf_dsrDataLinkInterface class. Next, the “StartSending” method is invoked on the path request.
- OnReceiveData – This method takes a byte array representing the received data, a S5066Address object representing the data source, and a boolean indicating whether the data was received ARQ or non-ARQ. First, the received data identified to determine what packet type it represents. Second, the appropriate packet class is created and passed the supplied byte array. Next, the appropriate data members are retrieved from the packet class and placed into the array of state machine parameters. Finally, the state machine is invoked differently based on packet type.
- ProcessAckPacket – This method takes a S5066Address object representing the packet source, an array of S5066Address objects representing the complete reverse path, an unsigned integer representing a unique identifier, and a boolean indicating whether the acknowledgment packet was received ARQ or non-ARQ. First, the reverse path is examined to determine if the local node needs to take action. If the local node exists on the reverse path, then the acknowledgment packet is sent to the next node in the reverse path. Otherwise, no action is taken.
- ProcessReversePathRequest – This method takes a S5066Address object representing the packet source, an array of S5066Address objects representing the complete forward path, a S5066Address representing the final destination for the reverse path request, an array of S5066Address objects representing the partial reverse path, and an unsigned integer representing a unique identifier. First, the “ShortenPath” method is invoked on the partial reverse path. Second, the local address is appended to the partial reverse path. Third, the “CalculateAndStoreRoutes” method is invoked on the forward path and the partial reverse path. Fourth, a determination is made whether or not to rebroadcast the reverse path request based on whether the local node already processed the unique identifier and if the local node is the final destination. If the local node did not process the unique identifier and is not the final destination, then the reverse path request is rebroadcast. Otherwise, no action is taken.
- ProcessPathRequest – This method takes a S5066Address object representing the packet source, an

array of S5066Address objects representing the partial forward path, a S5066Address representing the final destination, and an unsigned integer representing a unique identifier. First, the "ShortenPath" method is invoked on the partial forward path. Second, the local address is appended to the partial forward path. Third, a determination is made whether or not to rebroadcast the path request based on whether the local node already processed the unique identifier and if the local node is the final destination. If the local node did not process the unique identifier and is not the final destination, then the reverse path request is rebroadcast. If the local node did not process the unique identifier and is the final destination, then a reverse path request is generated and directed toward the original source (the first address in the forward path). Otherwise, no action is taken.

- ShortenPath – This method takes an array of S5066Address objects representing a path to be shortened. For every address A in the supplied path P, local route storage is searched for a shorter path P' from A to the local node. If one exists, it is returned. Otherwise, ShortenPath() returns P.
- ProcessDataPacket – This method takes a S5066Address representing the message source, an array of S5066Address objects representing the forward path, an unsigned integer representing the message size, an array of S5066Address objects representing the reverse path, a byte array representing a data segment, an unsigned integer representing the segment offset, an unsigned integer representing a unique identifier, and a boolean indicating whether the message was received ARQ or non-ARQ. First, the unique identifier is examined to determine if a partial message exists at the local node. If not, then one is created. Next, the received segment and segment offset are passed to the message. If the message is complete, then OnReceivedMessageComplete() is invoked.
- OnReceivedMessageComplete – This method takes a hf_dsrMessage object representing the complete message. If the local node is the message's final destination, then the OnMessageReceived() method is invoked on the session and an acknowledgment packet is directed toward the original message source. Otherwise, the StartSending() method is invoked on the message to forward it to the next recipient.
- SendFile – This method takes a S5066Address object indicating the file's final destination, a string containing the path to a file, and a boolean indicating whether the file should be sent using ARQ or non-ARQ delivery services. The three parameters are placed into the array of state machine parameters. Next, the state machine is invoked with the PSI_START_SENDING input.
- DiscoverPath – This method takes a S5066Address object representing the node for which routes are required. The parameter is placed into the array of state machine parameters. Next, the state machine is invoked with the PSI_DISCOVER_PATH input.
- Bind – This method takes a decimal representing the desired STANAG 5066 SAP to bind to, a S5066Address object representing our local individual address, and a S5066Address object representing our group address. First, the individual and group addresses are stored for local access. Next, the data link interface is invoked with the BindTo5066() method. The IP address to bind to is currently hardcoded to localhost on TCP port 5066.
- Unbind – First, this method resets the current state to "Idle". Next, the data link interface is invoked with the UnbindFrom5066() method.
- InvokeCurrentState – This method takes a ProtocolStateInput enumeration representing a state machine input. First, InvokeCurrentState() locks a mutex to protect the state machine methods from invocations on multiple threads. Second, the ProtocolStateInput is translated to a string and logged to the log file. Third, the state machine is invoked with the supplied input. Finally, the mutex is released.
- IdleState – This method represents the "Idle" state within the hf_dsrProtocol state machine. Six inputs are accepted in this state:
    - The PSI_START_SENDING handler extracts a S5066Address object representing a final destination node, a string representing a file path, and a boolean indicating whether the file should be sent ARQ from the array of state machine parameters. First, the handler searches for an existing route to and from the destination. If a route exists, then a new hf_dsrMessage is created, initialized with the route, and sent. Next, the current state is changed to "Waiting for Ack". Finally, the "Waiting for Ack" timer is started. If no route exists to the final destination, then the SendPathRequest() method is invoked, the current state is changed to "Waiting for Route to Send", and the "Waiting for Route" timer is started.
    - The PSI_DISCOVER_PATH handler extracts a S5066Address object representing a final destination node from the array of state machine parameters. The SendPathRequest() method is called, the "Waiting for Route" timer is started, and the current state is changed to "Waiting for Route".

        - The PSI_PATH_REQUEST_RECEIVED handler extracts a S5066Address object representing the path request sender, an array of S5066Address objects representing the nodes on the forward path so far, a S5066Address object representing the final destination node, and an unsigned integer representing a unique identifier from the array of state machine parameters. Next, the handler calls ProcessPathRequest() with the supplied parameters.
        - The PSI_REVERSE_PATH_REQUEST_RECEIVED handler extracts a S5066Address representing the reverse path request sender, an array of S5066Address objects representing the nodes on the forward path, a S5066Address object representing the original source node, an array of S5066Address objects representing the nodes on the reverse path so far, and an unsigned integer representing a unique identifier. Next, the handler calls ProcessReversePathRequest() with the supplied parameters.
        - The PSI_DATA_RECEIVED handler extracts a S5066Address object representing the data packet sender, an array of S5066Address objects representing the nodes in the forward path, an unsigned integer representing the total message size, an array of S5066Address objects representing the nodes in the reverse path, a byte array representing the received data segment, an unsigned integer representing the segment offset, an unsigned integer representing a unique identifier, and a boolean indicating whether the data was received ARQ or non-ARQ from the array of state machine parameters. Next, the handler calls ProcessDataPacket() with the supplied parameters.
        - The PSI_ACK_RECEIVED handler extracts a S5066Address object representing the acknowledgment packet sender, an array of S5066Address objects representing the nodes on the reverse path, an unsigned integer representing a unique identifier, and a boolean indicating whether the acknowledgment packet was received ARQ or non-ARQ from the array of state machine parameters. Next, the last node of the reverse path is compared with the local node address. If they match, then the local node is the acknowledgment packet's final destination. This should not occur in the “Idle” state. If they do not match, then the handler calls ProcessAckPacket() with the supplied parameters.
- WaitingForRouteToSend - This method represents the “Waiting for Route to Send” state within the hf_dsrProtocol state machine. Five inputs are accepted in this state:
    - The PSI_PATH_REQUEST_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_REVERSE_PATH_REQUEST_RECEIVED handler extracts the same objects from the array of state machine parameters as the corresponding handler in the IdleState() method. First, the handler invokes ProcessReversePathRequest() with the supplied parameters. Second, the original source address supplied by the reverse path request packet is compared against the local node address. If the two addresses do not match, then no action is taken. Otherwise, the route discovery is complete. CalculateAndStoreRoutes() is called on the newly discovered forward and reverse paths. Next, a new hf_dsrMessage is created. The message is sent to the next node on the forward path. Finally, the current state is changed to “Waiting for Ack”.
    - The PSI_DATA_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_ACK_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_WAITING_FOR_ROUTE_TIMER_EXPIRED handler changes the current state to “Idle”.
- WaitingForRoute – This method represents the “Waiting for Route” state within the hf_dsrProtocol state machine. Five inputs are handled in this state:
    - The PSI_PATH_REQUEST_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_REVERSE_PATH_REQUEST_RECEIVED handler extracts the same objects from the array of state machine parameters as the corresponding handler in the IdleState() method. First, the handler invokes ProcessReversePathRequest() with the supplied parameters. Second, the original source address supplied by the reverse path request packet is compared against the local node address. If the two addresses do not match, then no action is taken. Otherwise, the route discovery is complete. Next, CalculateAndStoreRoutes() is called on the newly discovered forward and reverse paths. Finally, the current state is changed to “Waiting for

Ack".
    - The PSI_DATA_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_ACK_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_WAITING_FOR_ROUTE_TIMER_EXPIRED handler behaves identically to the corresponding handler in the "Waiting for Route to Send" state.
- WaitingForAck – This method represents the "Waiting for Ack" state within the hf_dsrProtocol state machine. Five inputs are handled in this state:
    - The PSI_PATH_REQUEST_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_REVERSE_PATH_REQUEST_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_DATA_RECEIVED handler behaves identically to the corresponding handler in the IdleState() method.
    - The PSI_ACK_RECEIVED handler extracts the same objects from the array of state machine parameters as the corresponding handler in IdleState(). First, the last node of the reverse path is compared against the local address. If there is a match, then OnOutgoingMessageComplete() is called on the hf_dsrSession object and the current state is changed to "Idle". Otherwise, ProcessAckPacket() is invoked with the supplied parameters.
    - The PSI_WAITING_FOR_ACK_TIMER_EXPIRED_TIMER_EXPIRED handler changes the current state to "Idle".

4.1.1.1.1.5hf_dsrDataLinkInterface Class

The hf_dsrDataLinkInterface class encapsulates the STANAG 5066 interface for the benefit of the hf_dsrProtocol class. Functionality is exposed to send, bind, and unbind from the STANAG 5066 stack. Additionally, the hf_dsrDataLinkInterface class passes events to hf_dsrProtocol associated with incoming data. Finally, hf_dsrDataLinkInterface interfaces with the ThreadedSocketWrapper class to process outgoing data requests and receive incoming data requests.

The main functional requirement for hf_dsrDataLinkInterface involves encoding and decoding S_PRIMITIVEs. In this manner, STANAG 5066 details are abstracted away from much of the HF-DSR implementation. Additionally, hf_dsrDataLinkInterface handles asynchronous failure scenarios associated with binding to STANAG 5066.

A state machine is contained within hf_dsrDataLinkInterface that is constructed similarly to the state machine within hf_dsrSession. Two states are represented within the hf_dsrProtocol state machine: "Unbound" and "Bound".

The state machine handles the following inputs related to user initiated commands:

- SMI_BIND_REQUEST – This input represents a request to bind to STANAG 5066.
- SMI_UNBIND_REQUEST – This input represents a request to unbind from STANAG 5066.

The state machine handles the following inputs related received S_PRIMITIVEs:

- SMI_BIND_ACCEPT_RXD – This input represents an incoming S_BIND_ACCEPTED S_PRIMITIVE..
- SMI_BIND_REJECT_RXD – This input represents an incoming S_BIND_REJECTED S_PRIMITIVE..
- SMI_UNIDATA_INDICATION_RXD – This input represents an incoming S_UNIDATA_INDICATION S_PRIMITIVE..

The hf_dsrDataLinkInterface class contains the following methods:

- Constructor – The constructor takes a reference to an hf_dsrProtocol, which is stored locally. In this manner, data link interface events may be provided to the rest of HF-DSR. Next, the state machine is initialized. Additionally, an event is created to signal bind complete status between threads. Finally, a ThreadedSocketWrapper object is created for communication with a STANAG 5066 server.
- BindTo5066 – This method takes an IPAddress object representing the STANAG 5066 server host, an integer representing the STANAG 5066 server port, and a decimal representing the HF-DSR SAP ID from the array of state machine parameters. First, the SAP ID is stored for use during send operations. Next, the three parameters are assigned to the array of state machine parameters. The state machine is subsequently invoked with the SMI_BIND_REQUEST input. Next, the a "bind complete" event is invoked with WaitOne() and a delay of 5 seconds. This allows the STANAG 5066 server an opportunity to respond appropriate to the bind request. If WaitOne() returns 'true', then the STANAG 5066 server responded appropriately and 'true' is returned. Otherwise, 'false' is returned.
- UnbindFrom5066 – This method invokes the state machine with the SMI_UNBIND_REQUEST input.
- Send – This method takes a byte array representing the data to send, a S5066Address object representing the next node destination, a boolean indicating whether the data should be sent ARQ or non-ARQ, and an unsigned integer representing a transmission delay in milliseconds. If the transmission delay is greater than 0 milliseconds, then a timer is created and started for the appropriate period. Otherwise, the remaining parameters are assigned to the array of state machine parameters, and the state machine is invoked with the SMI_SEND_REQUEST input.
- OnTxDelayTimerExpired – This method takes an object containing the state machine parameters that would have been passed from Send() to the state machine if no transmission delay had been passed to Send(). OnTxDelayTimerExpired() assigns the supplied state machine parameters to the array, and invokes the state machine with the SMI_SEND_REQUEST input.
- OnReceiveData – This method takes a byte array representing the newly received data. First, the S5066S_PRIMITIVE.Identify() method is called with the received data as a parameter. Based on Identify()'s return value, different actions are taken. If Identify() returns BIND_ACCEPTED, then a new S5066BindAccept object is created and passed the received data. Next, the S5066BindAccept object is assigned to the array of state machine parameters. Finally, the state machine is invoked with the SMI_BIND_ACCEPT_RXD input. If Identify() returns BIND_REJECTED, no action is taken because the "bind complete" event times out and the bind process fails. If Identify() returns UNIDATA_INDICATION, then a new S5066UnidataIndication object is created and passed the received data. Next, the S5066UnidataIndication object is assigned to the array of state machine parameters. Finally, the state machine is invoked with the SMI_UNIDATA_INDICATION_RXD input.
- UnboundState – This method represents the "Unbound" state in the hf_dsrDataLinkInterface state machine. Two inputs are accepted in this state:
    - SMI_BIND_REQUEST – This handler extracts an IPAddress object representing the STANAG 5066 server address, an integer representing the STANAG 5066 server port, and a decimal representing the SAP to bind to from the array of state machine parameters. First, an IPEndPoint object is created with the IPAddress and port number. Next, The Connect() method is invoked on the SocketWrapper instance variable with the newly created IPEndPoint. If the SocketWrapper's "Connected" property is 'true', then a S5066BindRequest object is created with the supplied parameters and some hardcoded constants. The hardcoded constants include the STANAG 5066 rank (set to 1), the default service type (set to ARQ), the default delivery confirmation type (set to node confirmation), the default delivery order (set to ordered delivery), and the minimum number of non-ARQ retries (set to 1). Finally, the Send() method is invoked on the S5066BindRequest with the SocketWrapper reference.
    - SMI_BIND_ACCEPT – This handler sets the "Bind Completion" event.
- BoundState – This method represents the "Bound" state in the hf_dsrDataLinkInterface state machine. Three inputs are accepted in this state:
    - SMI_UNBIND_REQUEST – This handler creates a STANAG5066UnbindRequest object and invokes the Send() method on it. The Send() method is provided with a reference to a SocketWrapper object. Next, the SocketWrapper is invoked with the Close() method. Finally, the current state is changed to "Unbound".
    - SMI_SEND_REQUEST – This handler extracts a byte array representing the data to send, a S5066Address object representing the data destination, and a boolean indicating whether the data should be sent ARQ or non-ARQ from the array of state machine parameters. A

S5066UnidataRequest object is created, provided with the supplied parameters, and provided with the SAP identifier that HF-DSR bound with. Next, the S5066UnidataRequest object is invoked with the Send() method. The Send() method takes a reference to a SocketWrapper object.

- SMI_UNIDATA_INDICATION_RXD – This handler extracts a S5066UnidataIndication object from the array of state machine parameters. Next, the "TransMode" property is extracted from the S5066UnidataIndication. This indicates whether the data was received ARQ or non-ARQ. Finally, the hf_dsrProtocol reference is invoked with OnReceiveData(). OnReceiveData() takes the "U_PDU" property from the S5066UnidataIndication, the "SrcAddress" property from the S5066UnidataIndication, and a boolean indicating whether the data was received ARQ or non-ARQ.

4.1.1.1.1.6IThreadedSocketWrapperEvents Interface

The IThreadedSocketWrapper interface is used to handle received data notifications from ThreadedSocketWrapper. It is implemented by the hf_dsrDataLinkInterface class.

The IThreadedSocketWrapperEvents contains the following method:

- OnReceiveData - This method takes a byte array representing the newly received data. Any actions taken are dependent on the overriding class.

4.1.1.1.1.7ThreadedSocketWrapper Class

The ThreadedSocketWrapper class provides an abstraction layer between HF-DSR and the TCP/IP connection to the STANAG 5066 server. ThreadedSocketWrapper allows user-initiated requests to complete immediately while the requested operation is still pending. Additionally, incoming data is processed on independent threads. In this manner, received data is analyzed more quickly than it would be on the user-interface thread.

Incoming requests for the following actions can take place on any thread:

- Send Request – Send requests cause blocks of data to be transferred to the TCP/IP peer.
- Close Request – Close requests terminate the TCP/IP connection with the peer.

The internal ThreadedSocketWrapper.QueuedRequest class represents requests from arbitrary threads to perform "Send" and "Close" requests. These types of requests generate a QueuedRequest object, add it to a queue, and set an event. A worker thread stops waiting for the event, dequeues all QueuedRequest objects, acts on them, and resumes waiting on the event.

The internal ThreadedSocketWrapper.QueuedSendRequest and ThreadedSocketWrapper.QueuedCloseRequest classes inherit from QueuedRequest. These classes represent "Send" and "Close" requests, respectively.

The .NET mechanism for asynchronous receive operations is utilized to ensure that receive operations take place independently of other threads. The method that initializes asynchronous receives returns immediately. A callback method is invoked by .NET when receive completes. This callback occurs on a .NET managed thread.

The ThreadedSocketWrapper class contains the following methods:

- Constructor – The constructor takes a IThreadedSocketWrapper reference as a parameter. First, the parameter is stored for an event sink. Next, the internal Socket object, request queue, new request event, quit event, and received data buffer is initialized.
- Send – This method takes a byte array representing the data to send as a parameter. First, a QueuedSendRequest object is created with the byte array. Next, the object is passed to the queue.

Finally, the new request event is set.

- Close – First, this method creates a QueuedCloseRequest. Next, the object is passed to the queue. Finally, the new request event is set.
- Connect – This method takes an EndPoint object representing the STANAG 5066 server as a parameter. First, an AutoResetEvent object is created. Next, a new Socket object is created. The BeginConnect() method is invoked on the new socket and passed the EndPoint object, the "OnConnectComplete" callback method, and the new event. Finally, the WaitOne() method is invoked on the event. This causes Connect() to block until the connection attempt is complete.
- OnConnectComplete – This method takes an IAsyncResult reference representing the result of the connection attempt as a parameter. First, the AutoResetEvent provided in the Connect() method is extracted from the IAsyncResult. Next, the EndConnect() method is invoked on the Socket object involved in the connection. If EndConnect() throws a SocketException, then the connection attempt has failed. Otherwise, the BeginReceive() method is invoked on the Socket object. This causes the asynchronous receive operation to start. Next, the worker thread responsible for dequeuing QueuedRequests is started. Finally, the AutoResetEvent provided in the Connect() method is set, which causes the Connect() method to return. This occurs regardless of whether EndConnect() threw an exception.
- QueuedRequestThreadProc – This method is executed on the worker thread responsible for dequeuing QueuedRequest objects. First, QueuedRequestThreadProc() waits on the "New Request" and "Quit" events. If the "Quit" event is signaled, then QueuedRequestThreadProc() returns. If the "New Request" event is signaled, then the type of QueuedRequest is determined. If the QueuedRequest is a "Send" request, then the data is extracted from the QueuedRequest and sent to STANAG 5066. If the QueuedRequest is a "Close" request, then the socket is shut down and QueuedRequestThreadProc returns, ending its thread.
- OnReceiveComplete – This method is called in response to data being received on the internal Socket object. Data is extracted from the Socket object and passed to the hf_dsrDataLinkInterface for processing. Next, BeginReceived() is invoked on the Socket object. In this manner, additional receive operations cause OnReceiveComplete() to be executed again. Any exceptions thrown while invoking OnReceiveComplete() causes the QueuedRequest thread to quit.

4.1.1.1.1.8TimestampTraceListener Class

The TimestampTraceListener class inherits from the .NET TextWriterTraceListener class. It is a diagnostic tool that registers with the .NET Trace object. Once registered, all strings logged to the Trace object are passed to the TimestampTraceListener for processing.

TimestampTraceListener operates nearly identically as TextWriterTraceListener. Text strings logged to the Trace object are logged to a text file on the file system. Additionally, TimestampTraceListener prepends the date and time to each string that is logged.

The TimestampTraceListener class contains the following methods:

- Constructor – The constructor takes a Stream object that represents an output mechanism. The only action taken by the constructor is to invoke the base class constructor. The base class constructor is passed the supplied Stream object.
- WriteLine – The WriteLine method takes a string representing the text to be logged. First, the current date and time are prepended to the supplied string. Next, the string is supplied to the base class WriteLine() implementation.

4.1.1.1.1.9hf_dsrPacket Class

The hf_dsrPacket class is the base class for all HF-DSR level packets. It is an abstract class that defines methods to be overridden, provides functionality common to all packets, and defines data types common to all packet subclasses.

The hf_dsrPacket class contains the following methods:

- Constructor – The constructor takes no parameters and performs no tasks.
- Identify – This static method takes an array of bytes that represent an hf_dsrPacket. First, Identify() determines what type of hf_dsrPacket the byte array represents. Next, the specific type is returned as a member of the PACKET_TYPES enumeration. In this manner, clients can determine which subclass of hf_dsrPacket can be instantiated with the byte array.
- StartSending – This abstract method takes an unsigned integer representing the transmission delay in milliseconds. hf_dsrPacket subclasses override this method in order to send themselves.

4.1.1.1.1.10hf_dsrAck Class

The hf_dsrAck class inherits from hf_dsrPacket and provides functionality to encode and decode acknowledgment packets.

The hf_dsrAck class contains the following methods:

- Constructor – This constructor takes a byte array representing a serialized hf_dsrAck object. Typically, the hf_dsrAck object is serialized when newly received from STANAG 5066. The constructor de-serializes the byte array and uses its contents to initialize instance variables.
- Constructor – This constructor takes an array of S5066Address objects representing the reverse path, an unsigned integer representing a unique identifier, a S5066Address object representing the next destination for the hf_dsrAck packet, a boolean indicating whether the acknowledgment packet should be sent ARQ or non-ARQ, and a reference to the application's hf_dsrDataLinkInterface object. The constructor initializes instance variables based on the supplied parameters.
- StartSending – This method takes an unsigned integer representing the transmission delay in milliseconds. StartSending() serializes the object instance data into a byte array conforming to the abovementioned HF-DSR Acknowledgment packet format. Next, the byte array, a S5066Address object representing the next node destination, a boolean representing ARQ or non-ARQ, and the transmission is passed to hf_dsrDataLinkInterface.Send().

4.1.1.1.1.11hf_dsrData

The hf_dsrData class inherits from hf_dsrPacket and provides functionality to encode and decode data packets.

The hf_dsrData class contains the following methods:

- Constructor – This constructor takes a byte array representing a serialized hf_dsrData object. Typically, the hf_dsrData object is serialized when newly received from STANAG 5066. The constructor de-serializes the byte array and uses its contents to initialize instance variables.
- Constructor – This constructor takes an array of S5066Address objects representing the forward path, an array of S5066Address objects representing the reverse path, a byte array representing the data segment to send, an unsigned integer representing the segment offset, an unsigned integer representing the total message size, an unsigned integer representing a unique identifier, a S5066Address object representing the next destination for the hf_dsrData packet, a boolean indicating whether the acknowledgment packet should be sent ARQ or non-ARQ, and a reference to the application's hf_dsrDataLinkInterface object. The constructor initializes instance variables based on the supplied parameters.
- StartSending – This method takes an unsigned integer representing the transmission delay in milliseconds. StartSending() serializes the object instance data into a byte array conforming to the abovementioned HF-DSR Data packet format. Next, the byte array, a S5066Address object representing the next node destination, a boolean representing ARQ or non-ARQ, and the transmission is passed to hf_dsrDataLinkInterface.Send().

4.1.1.1.1.12hf_dsrPathRequest

The hf_dsrPathRequest class inherits from hf_dsrPacket and provides functionality to encode and decode

Path Request packets.

The hf_dsrPathRequest class contains the following methods:

- Constructor – This constructor takes a byte array representing a serialized hf_dsrPathRequest object. Typically, the hf_dsrPathRequest object is serialized when newly received from STANAG 5066. The constructor de-serializes the byte array and uses its contents to initialize instance variables.
- Constructor – This constructor takes a S5066Address object representing the node to discover, a S5066Address object representing the network's group address, an array of S5066Address objects representing the accumulated forward path so far, an unsigned integer representing a unique identifier, and a reference to the application's hf_dsrDataLinkInterface object. The constructor initializes instance variables based on the supplied parameters.
- StartSending – This method takes an unsigned integer representing the transmission delay in milliseconds. StartSending() serializes the object instance data into a byte array conforming to the abovementioned HF-DSR Path Request packet format. Next, the byte array is supplied to the hf_dsrDataLinkInterface.Send() method. Next, the byte array, a S5066Address object representing the next node destination, a boolean representing ARQ or non-ARQ, and the transmission is passed to hf_dsrDataLinkInterface.Send().

4.1.1.1.1.13hf_dsrReversePathRequest

The hf_dsrReversePathRequest class inherits from hf_dsrPacket and provides functionality to encode and decode Reverse Path Request packets.

The hf_dsrReversePathRequest class contains the following methods:

- Constructor – This constructor takes a byte array representing a serialized hf_dsrReversePathRequest object. Typically, the hf_dsrReversePathRequest object is serialized when newly received from STANAG 5066. The constructor de-serializes the byte array and uses its contents to initialize instance variables.
- Constructor – This constructor takes a S5066Address object representing the original source node to discover, a S5066Address object representing the network's group address, an array of S5066Address objects representing the accumulated reverse path so far, an array of S5066Address objects representing the forward path, an unsigned integer representing a unique identifier, and a reference to the application's hf_dsrDataLinkInterface object. The constructor initializes instance variables based on the supplied parameters.
- StartSending – This method takes an unsigned integer representing the transmission delay in milliseconds. StartSending() serializes the object instance data into a byte array conforming to the abovementioned HF-DSR Reverse Path Request packet format. Next, the byte array, a S5066Address object representing the next node destination, a boolean representing ARQ or non-ARQ, and the transmission is passed to hf_dsrDataLinkInterface.Send().

4.1.1.1.1.14hf_dsrMessage

The hf_dsrMessage class represents a file to be sent. hf_dsrMessage contains functionality to packetize and de-packetize the contents of a file. Upon creation, the designated file is opened and its contents are read into a stream. The filename is appended to the file contents, and the file is segmented into evenly-sized segments. Each segment is passed to an hf_dsrData packet and sent to STANAG 5066.

hf_dsrData packets received from STANAG 5066 are passed into the hf_dsrMessage object that corresponds to their received message. Once all segments associated with an incoming hf_dsrMessage are received, the segment is re-transmitted as directed by the hf_dsrMessage client.

The hf_dsrMessage class contains the following methods:

- Constructor – This constructor takes a string representing a filename, an array of S5066Address objects representing the forward path, an array of S5066Address objects representing the reverse path, an

unsigned integer representing a unique identifier, a S5066Address object representing the next node destination, a boolean representing whether the message should be transferred in ARQ mode, and the application's reference to the hf_dsrDataLinkInterface object. This particular constructor is called by the original message sender. The constructor initializes instance data with the supplied parameters.

- Constructor – This constructor takes an array of S5066Address objects representing the forward path, an array of S5066Address objects representing the reverse path, an unsigned integer representing the total message size, an unsigned integer representing a unique identifier, a S5066Address object representing the next node destination, a boolean indicating whether the message should be transferred in ARQ mode, and a reference to the application's hf_dsrDataLinkInterface object. This constructor is called by intermediate and final message receivers before any message contents have been deposited into the object. The method implementation initializes the message contents stream to an empty MemoryStream object and initializes instance data in accordance with the supplied parameters.
- StartSending – This method takes no parameters. First, a BinaryReader object is opened on the message contents stream. Next, constant sized data quantities are extracted from the BinaryReader and placed into hf_dsrData objects. Finally, the hf_dsrData objects are submitted to STANAG 5066.
- OnDataPacketRxd – This method takes a byte array representing the received message segment and an unsigned integer representing the segment offset into the message. The segment is written into the appropriate position within the contents stream.
- LoadFilenameFromContents – This method takes no parameters. First, the seek pointer on the contents stream is placed at its beginning. Next, two byte quantities are read from the contents stream and typecast into a single character. The character is appended to a filename until the NULL (value = 0) character is encountered.

4.1.1.1.1.15S5066S_Primitive Class

The S5066S_Primitive class is the base class for all S_PRIMITIVE types implemented with HF-DSR. S5066S_Primitive contains functionality common to all S_PRIMITIVEs. Specifically, S5066S_Primitive can encode and decode the common S_PRIMITIVE header. As a result, incoming S_PRIMITIVE types may be identified during decoding so that the correct S5066S_Primitive subclass can be instantiated.

S5066S_Primitive implements the following methods:

- Constructor – This constructor takes an unsigned short representing the number of bytes in the specific (non-header) portion of the S_PRIMITIVE. This constructor is typically called when creating a S5066S_Primitive for transfer to STANAG 5066. The constructor stores the parameter as a member variable and returns.
- Constructor – This constructor takes a byte array representing a serialized S_PRIMITIVE. This constructor is typically called when creating a S5066S_Primitive from received STANAG 5066 data. The constructor accesses the proper offset into the byte array to retrieve the number of bytes in the non-header portion of the S_PRIMITIVE.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. Send() is an abstract method intending to be overridden by its subclasses. Overridden implementations for Send() are intended to submit a serialized version of the proper S_PRIMITIVE to the supplied ThreadedSocketWrapper object.
- Identify – This method takes a byte array representing a serialized S_PRIMITIVE. Identify() is a static method that does not require S5066S_Primitive object instantiation to invoke. Instead, clients call IdentifyO to determine the specific type of S5066S_Primitive subclass to create based on received STANAG 5066 data. Identify() accesses the appropriate offset of the supplied byte array to determine the specific S_PRIMITIVE type. The resulting type enumeration is returned.
- Encode – This method takes a byte array representing the location in which to encode data. Encode is typically called from S5066S_Primitive subclasses Encode() implementations. If the supplied array is large enough, Encode() serializes the common S_PRIMITIVE header into the byte array and returns “true”. Otherwise, Encode() returns “false”.

4.1.1.1.1.16S5066BindAccept Class

S5066BindAccept is a subclass of S5066S_Primitive and contains functionality to decode

S_BIND_ACCEPTED S_PRIMITIVEs. S5066BindAccept is not capable of encoding the S_BIND_ACCEPTED S_PRIMITIVE because HF-DSR never needs to encode it.

S5066BindAccept contains the following methods:

- Constructor – The constructor takes a byte array representing a serialized S_BIND_ACCEPT S_PRIMITIVE. First, the base class S5066S_Primitive constructor is called and passed the supplied byte array. Next, the specific S5066BindAccept fields are extracted from the appropriate portion of the byte array.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. Send() is overridden because the compiler requires subclasses of S5066S_Primitive to override the abstract S5066S_Primitive.Send() method. However, S5066BindAccept.Send() performs no tasks and returns immediately.

4.1.1.1.1.17S5066BindRequest Class

S5066BindRequest is a subclass of S5066S_Primitive and contains functionality to encode S_BIND_REQUEST S_PRIMITIVEs. S5066BindRequest is not capable of decoding incoming S_BIND_REQUEST S_PRIMTIIVEs because HF-DSR never needs to decode them.

S5066BindRequest contains the following methods:

- Constructor – This constructor takes an integer representing a SAP identifier, an integer representing the binding client rank, a boolean representing if the client wishes to bind in ARQ/non-ARQ or non-ARQ only mode, a DEL_CONF enumeration selection indicating the default type of delivery confirmation, a boolean representing whether the client wishes to bind with ordered delivery enforced, and an integer representing the minimum number of non-ARQ packet repeats. First, the base class constructor is called and passed the number of bytes in the S_BIND_REQUEST non-header portion. Next, the parameters are all stored as member variables.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. First, Send() calls the base class implementation of Send(), which causes the common S_PRIMITIVE header to be written to a byte array. Next, Send() causes the S5066BindRequest object's internal state to be serialized into the byte array after the header. Finally, the byte array is submitted to the ThreadedSocketWrapper for transfer to STANAG 5066.

4.1.1.1.1.18S5066UnbindRequest Class

The S5066UnbindRequest class is a subclass of S5066S_Primitive and contains functionality to encode S_UNBIND_REQUEST S_PRIMITIVEs. S5066UnbindRequest is not capable of decoding incoming S_UNBIND_REQUEST S_PRIMTIIVEs because HF-DSR never needs to decode them.

S5066UnbindRequest contains the following methods:

- Constructor – This constructor contains no arguments. The base class constructor is called and passed the number of bytes in the S_UNBIND_REQUEST non-header portion.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. First, Send() calls the base class implementation of Send(), which causes the common S_PRIMITIVE header to be written to a byte array. Next, Send() causes the S5066UnbindRequest object's internal state to be serialized into the byte array after the header. Finally, the byte array is submitted to the ThreadedSocketWrapper for transfer to STANAG 5066.

4.1.1.1.1.19S5066UnidataIndication Class

S5066UnidataIndication is a subclass of S5066S_Primitive and contains functionality to decode S_UNIDATA_INDICATION S_PRIMITIVEs. S5066UnidataIndication is not capable of encoding the S_UNIDATA_INDICATION S_PRIMITIVE because HF-DSR never needs to encode it.

S5066UnidataIndication contains the following methods:

- Constructor – The constructor takes a byte array representing a serialized S_UNIDATA_INDICATION S_PRIMITIVE. First, the base class S5066S_Primitive constructor is called and passed the supplied byte array. Next, the specific S5066UnidataIndication fields are extracted from the appropriate portion of the byte array.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. Send() is overridden because the compiler requires subclasses of S5066S_Primitive to override the abstract S5066S_Primitive.Send() method. However, S5066UnidataIndication.Send() performs no tasks and returns immediately.

4.1.1.1.1.20S5066UnidataRequest Class

The S5066UnidataRequest class is a subclass of S5066S_Primitive and contains functionality to encode S_UNIDATA_REQUEST S_PRIMITIVEs. S5066UnidataRequest is not capable of decoding incoming S_UNIDATA_REQUEST S_PRIMTIIVEs because HF-DSR never needs to decode them.

S5066UnidataRequest contains the following methods:

- Constructor – This constructor contains no arguments. The base class constructor is called and passed the number of bytes in the S_UNIDATA_REQUEST non-header portion.
- Send – This method takes a ThreadedSocketWrapper object reference representing the connection to STANAG 5066. First, Send() calls the base class implementation of Send(), which causes the common S_PRIMITIVE header to be written to a byte array. Next, Send() causes the S5066UnidataRequest object's internal state to be serialized into the byte array after the header. Finally, the byte array is submitted to the ThreadedSocketWrapper for transfer to STANAG 5066.

5 User Manual

This section represents a user manual for the HF-DSR implementation developed for this project.

5.1.1.1Minimum System Requirements

Please note that HF-DSR may operate successfully even if one or more minimum requirements are not met. The following are the minimum requirements tested:

- Microsoft Windows XP
- 450MHz Intel Pentium 2 Compatible CPU
- 256MB memory
- 2 unused 9-pin RS-232 (COM) ports. USB to COM port interfaces may be used if insufficient COM ports are available.
- Microsoft .NET Framework v1.1
- Harris RF-6710W Wireless Messenging Terminal or RF-6750W Wireless Gateway (RF-67x0W)
- MIL-STD-188-141A compliant ALE-capable radio controlled by RF-67x0W (refer to RF-67x0W documentation)
- Radio-embedded or external HF modem capable of:
    - MIL-STD-188-141 “Serial Tone” operation
    - Remote control from RF-67x0W (refer to RF-67x0W documentation)
- Harris Synchronous Communication Interface Card
- All necessary radio and modem cables

5.1.1.2Hardware Setup

As discussed in the RF-67x0W user manual, perform the following steps:

- Install the Harris Synchronous Communication Interface Card onto the appropriate bus (PCI slot, PCMCIA slot, ISA slot)
- Connect the 25 pin connector on the Harris Synchronous Communication Interface Card to the HF modem. Use a cable provided with the modem or radio to make this connection.
- Connect one of the unused COM ports to the radio using a cable provided with the radio. This connection is used to perform ALE (and possibly modem) control.
- If necessary, connect the other unused COM port to the radio or modem by using a cable provided with the radio or modem. This connection is used to perform modem control.
- Ensure that the radio is powered on and set to "Remote" mode, if applicable.
- If using an external modem, ensure it is powered on and set to "Remote" mode, if applicable.

5.1.1.3Software Setup

5.1.1.3.1.1RF-67x0W Configuration

As discussed in the RF-67x0W user manual, perform the following steps:

- Install the RF-67x0W software from the installation media.
- Run the Harris Synchronous Communication Interface Card Installation Utility from the RF-67x0W installation media.
- Configure one or more STANAG 5066 radio networks containing radios which participate in HF-DSR. Select "5066 Over Serial 110A (Sync DTE)" for the network type. This enables STANAG 5066 operation using the MIL-STD-188-110A modem waveform. Select "Enable S5066 IP" for those radio networks. This enables the socket-based S_PRIMITIVE interface. Disregard any default values generated for the "IP" column in the grid as they are not applicable to HF-DSR operation.

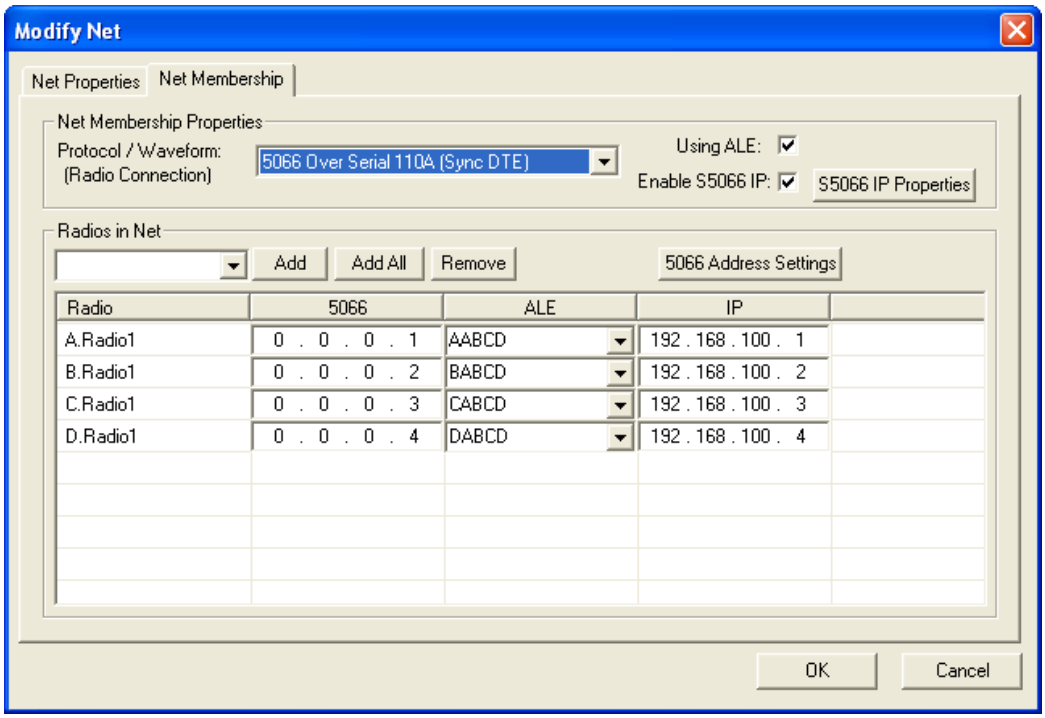

Figure 21: RF-67x0W Network Configuration page

5.1.1.3.1.2Microsoft .NET Framework Installation

Install the Microsoft .NET Framework from local media or the "Windows Update" web site at http://windowsupdate.microsoft.com.

5.1.1.3.1.3HF-DSR Installation

Copy HF-DSR.EXE onto the local hard drive. In order to obtain HF-DSR.EXE, follow the compilation instructions in section 5.4 below.

5.1.1.4Compilation

Install Microsoft Visual Studio .NET 2003. Within Visual Studio, open the "HF-DSR.csproj" project file. Under the "Build" menu, select "Configuration Manager" and ensure that the active solution configuration is "Release". Next, select "Build Solution" under the "Build" menu. Once the build process succeeds, HF-DSR.EXE is generated in the "bin/Release" directory

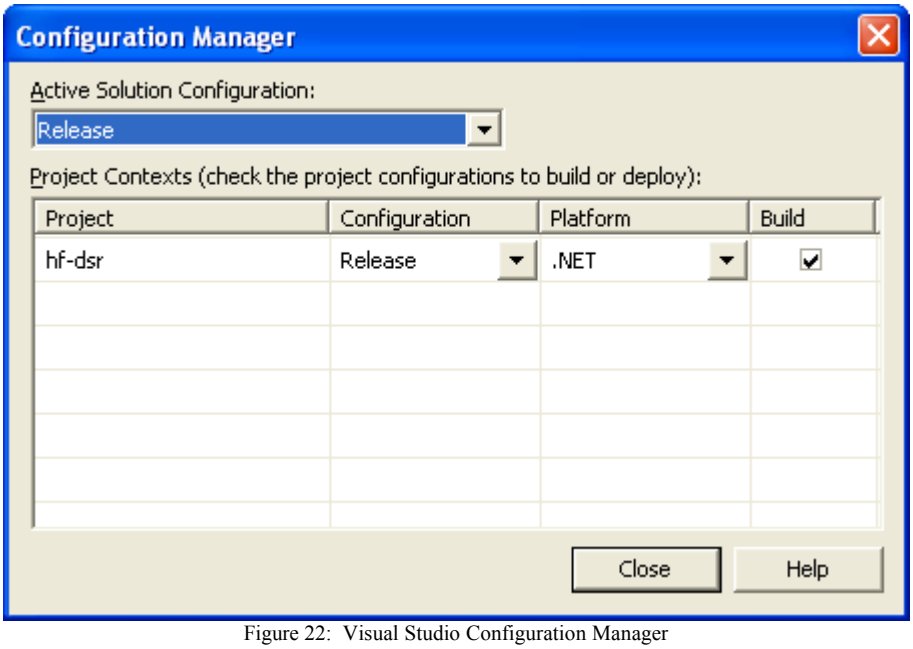

Figure 22: Visual Studio Configuration Manager

.

5.1.1.5HF-DSR Operation

Before starting HF-DSR.EXE, ensure that RF-67X0W is running and in "on-line" mode. Run the executable. After starting HF-DSR, certain configuration values are immediately accessible. The SAP ID, a client identifier for STANAG 5066, must be set between 0 and 15. Any values may be used except for 3, 9, and 12 because these values are reserved for other clients within RF-67x0W. All HF-DSR nodes must use the same SAP ID for successful communication.

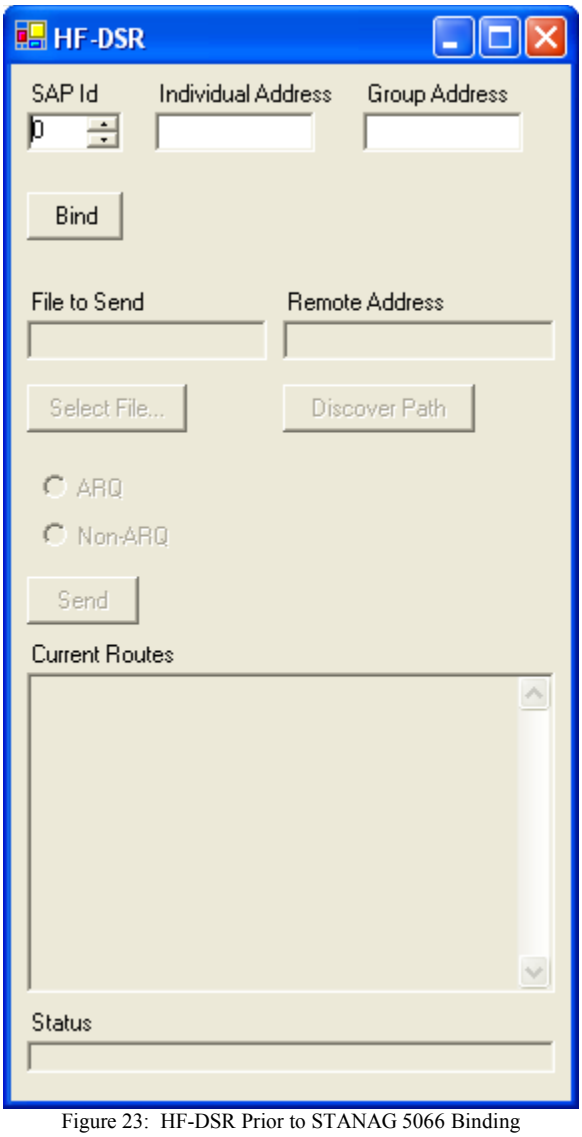

Figure 23: HF-DSR Prior to STANAG 5066 Binding

The “Individual Address” field must be set with the local node's STANAG 5066 address, which takes the form “A.B.C.D”. Using this representation, “A” may range from 0-15. “B” and “C” may range from 0-255 and “D” may range from 1-255. The address entered for this field must remain consistent with the local node's STANAG 5066 address within RF-67x0W.

The “STANAG 5066 Groupcast Address” within RF-67x0W is 31.255.255.255 by default. This address is configurable from the “Advanced Station Settings” property page. Note when entering the “Group Address” in HF-DSR, the high order bit must be removed from the groupcast address configured in RF-67x0W. For the default group address, HF-DSR must be configured with “15.255.255.255”.

Once these values are configured, click on the “Bind” button. This causes a connection attempt to STANAG 5066. If the attempt fails, the user is notified. If the attempt succeeds, the rest of the controls enable.

File paths may be entered within the “File to Send” field using the “Select File” button, which invokes a standard Windows file dialog. This field is only used when transferring a file, and is not necessary when invoking path discovery separately.

The “ARQ” and “non-ARQ” radio buttons determine the file transfer mode only. Route discovery always uses non-ARQ traffic.

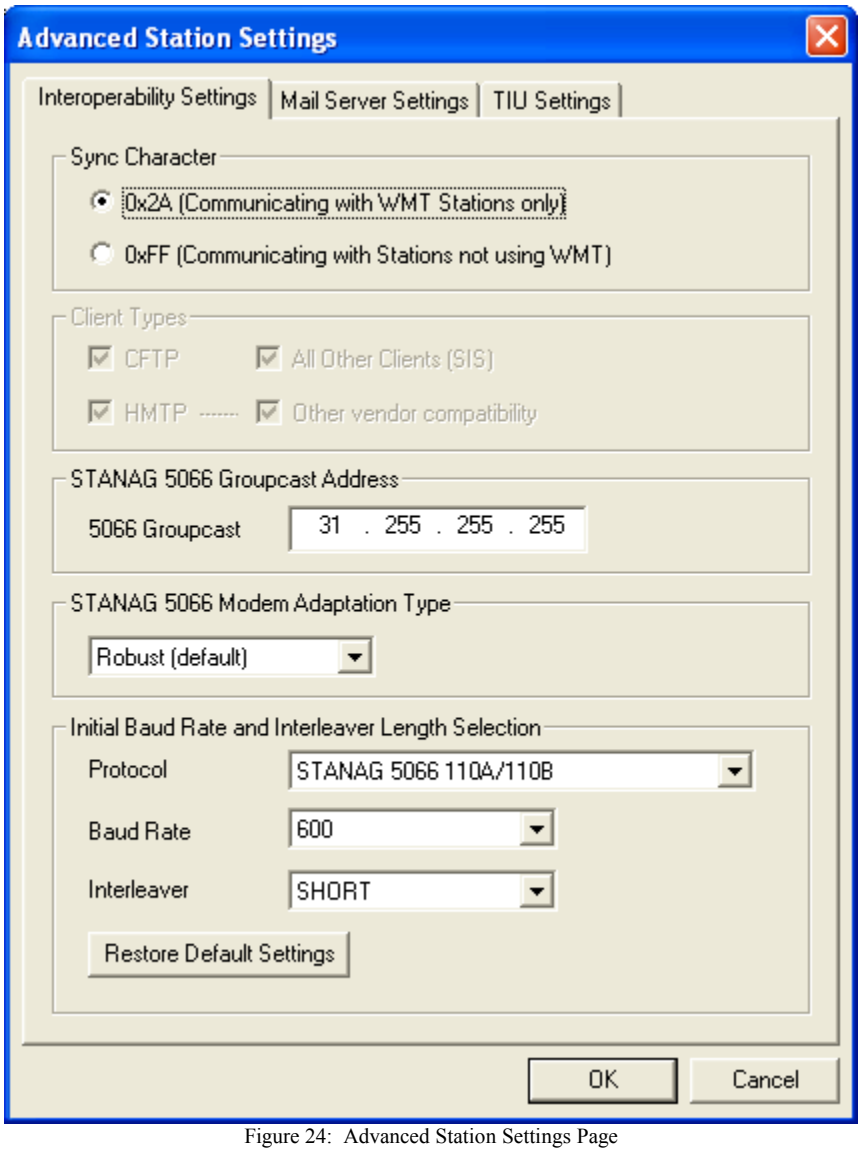

Figure 24: Advanced Station Settings Page

The “Discover Path” button initiates path discovery to a destination node specified in the “Remote Address” field. Path discovery is a potentially lengthy process and involves all nodes that have direct or indirect connectivity with the local node. There are two responses to a path discovery attempt. One possibility is that a Reverse Path Request is received and the discovered paths are displayed in the “Current Routes” area. The other possibility is that route discovery times out. In this case, the user is notified by an appropriate message in the “Status” area.

It is expected that HF-DSR participates in route discovery and file transfer initiated by other nodes. As such, nodes transmit regardless of whether they have a route discovery in progress, a file transfer in progress, or are otherwise idle.

The “Send” button queues the file selected in the “File to Send” for transmission pending a route to the node specified in the “Remote Address” area. Path discovery is automatically invoked if a route to the destination does not yet exist. In this case, path discovery proceeds as if the user clicked on the “Discover Path” button. Once a route is available, the file is sent to STANAG 5066 for transfer to the first node along the forward path. At this point, there are two possibilities. The first possibility is that an acknowledgment is received along the reverse path. The other possibility is that the file transfer times out. In either case, an appropriate message is displayed in the “Status” area.

6 HF-DSR Test Results

6.1.1.1Equipment Configuration

The HF-DSR implementation was exercised on four desktop PCs. Each PC was running a Harris RF-6710W Wireless Messaging Terminal containing a STANAG 5066 implementation. Additionally, each PC contained a Quatech PCI or PCMCIA synchronous RS-232 interface. The RS-232 interfaces were connected to HF modems, each of which was connected to an external or integrated tactical radio.

Modem types being used were Harris RF-5710A external modems, RF-5022E internal modems, and RF-5800H internal modems. The internal modems reside in the same chassis as the radio with the specified model number. The external modem connects to an arbitrary radio by using an audio connector.

Radio types being used were RF-5022, RF-5022E, and RF-5800H radios. RF-5022 radios were used with Harris RF-5710A external modems. The RF-5022E and RF-5800H radios used internal modems. The radios' RF outputs were directly connected to one another with co-axial cable. In order to prevent relatively powerful radio outputs from overloading other radios' inputs, each radio output signal was attenuated by 20 decibels. Additionally, each radio was configured to output 1 watt of power or less.

The radio network used for data collection is completely connected. As a result, radios within the network can receive signal emitted from any other radio in the network. Although this condition is not desirable when testing HF-DSR, it is unavoidable without additional switching equipment. In order to test partial network connectivity, ALE is used to prevent certain stations from contacting other stations. In this manner, distinct forward and reverse paths are possible.

Constructing ALE configurations to restrict radio connectivity involves designating different channel groups to be assigned to subsets of the radio network. Nodes scanning channel groups with common channels have connectivity. Conversely, nodes scanning channel groups with no common channels have no connectivity. Note that connectivity restricted in this fashion is bi-directional.

Unidirecitonal communication is not possible when using ALE to restrict connectivity in a test network. In order to accomplish this, it would be necessary for a node N to call a remote nodes R on a channel that is not part of N's scan list. ALE does not permit this.

Four nodes were available for testing. Although additional nodes would increase the quantity of network types that could be tested, the required equipment was not available. The nodes being used were configured as follows:

Node A:

- 3.2 ghz Pentium 4 Extreme Edition with 1GB RAM
- RF-5800H radio with internal modem
- Harris PCI to Synchronous RS-232 interface

Node B:

- 2.4 ghz Pentium 4 with 512MB RAM
- RF-5022E with internal RF-5285 modem
- Harris PCMCIA to Synchronous RS-232 interface

Node C:

- 450 mhz Pentium 2 with 256MB RAM
- RF-5022E with internal RF-5285 modem
- Harris PCI to Synchronous RS-232 interface

Node D

- 1.7 ghz Pentium 4 with 512MB RAM
- RF-5022 with external RF-5710A modem
- Harris PCI to Synchronous RS-232 interface

Each node used RF-6710W Wireless Messenging Terminal and Windows XP Professional.

6.1.1.2Test Configuration

The two network configurations being tested are a "linear" network and a "diamond" network. The linear network consists of four nodes designated as N_0 through N_3. In this network, N_i has connectivity with N_{i+1} and N_i has connectivity with N_{i-1}. N_0 and N_3 do not have connectivity with one another. HF-DSR's expected behavior within such a network is for path discovery traffic propagation to proceed from the originating node toward each network endpoint. If the path discovery destination is found along the path from the originator to the network endpoints, then the destination does not continue forwarding path requests. The expected behavior of reverse path request propagation throughout the network is identical, except that they originate from the path discovery target.

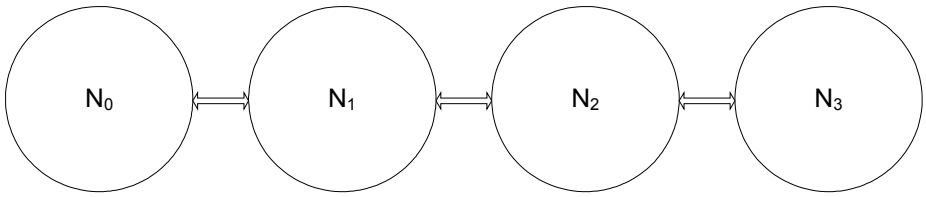

Figure 25: Linear Network

Diamond networks also consist of nodes N_0 through N_3. N_0 has connectivity with N_1 and N_2. Additionally, N_3 has connectivity with N_1 and N_2. Finally, N_1 and N_2 have connectivity with one another. As a result, path discovery traffic propagation is expected to operated differently than in a linear network. HF-DSR's expected behavior is for path discovery requests originated at N_0 or N_3 to be received by N_1 and N_2 simultaneously. Assuming that the opposite network endpoint is the intended destination, N_1 and N_2 attempt to rebroadcast the path request simultaneously. At this point, the level 2 protocols (STANAG 5066 and ALE) are responsible for arbitrating potential traffic collisions. The intended destination processes the forwarded path request from N_1 or N_2. The other path request is discarded. In this manner, the forward path constructed varies based on network conditions. The expected behavior of reverse path requests is similar in this scenario. Reverse paths contain either N_1 or N_2 depending on network conditions.

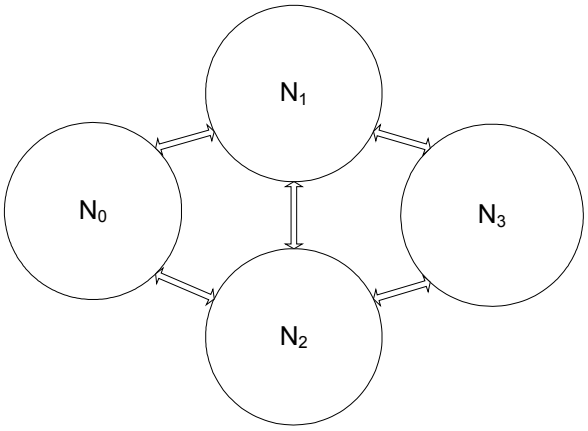

Figure 26: Diamond Network

The time required for path discovery and data transfer, and whether they are successful, is recorded for each network configuration.

Data transfer is exercised by using a 50K pre-compressed file. In this manner, no performance improvements are possible as a result of reduced file size during data compression.

6.1.1.3Test Execution

Test execution proceeded successfully except for two problems. The data transfer time limit was hard-coded to 10 minutes. As a result, data transfers that require in excess of 10 minutes timed out. In response, the maximum data transfer timeout was increased to one hour.

When testing diamond nets, it became apparent that over-the-air collisions were occasionally causing path discovery to fail. Data collisions involving the intended path discovery target caused interference with the first reverse path request transmission. As a result, a 20 second timer transmit delay timer was prepended to path-discovery transmissions from all other nodes. The intended path discovery target had no transmit delay timer. As a result, the path discovery target has precedence over other path discovery transmissions.

Data collisions involving intermediate nodes during path discovery did not necessarily cause failure. This behavior more often caused one transmission to be received and one to be lost. In the diamond network, this caused some degree of variance between discovered paths.

6.1.1.3.1.1Linear Networks

Path discovery within linear networks yielded the following results:

Source Node	Destination Node	Time Elapsed (in seconds)
N_0	N_1	53
N_0	N_2	124
N_0	N_3	224

Data transfer within linear networks yielded the following results:

Source Node	Destination Node	Time Elapsed (in seconds)
N_0	N_1	201
N_0	N_2	798
N_0	N_3	1145

6.1.1.3.1.2Diamond Networks

Path discovery within diamond networks yielded the following results:

Source Node	Destination Node	Time Elapsed (in seconds)
N_0	N_1	65
N_0	N_2	64
N_0	N_3	143

Data transfer within diamond networks yielded the following results:

Source Node	Destination Node	Time Elapsed (in seconds)
N_0	N_1	274
N_0	N_2	344
N_0	N_3	678

6.1.1.4Analysis

6.1.1.4.1.1Linear Networks

Path discovery proceeded as expected within linear networks. Path requests propagated from the initiator, at one endpoint of the network, until the intended destination was reached. At this point, the intended destination transmitted a reverse path request. In one direction, the reverse path request message terminated when the edge of the network was reached. In the other direction, the reverse path request message reached the original source.

Data transfer operated successfully within linear networks. Each node along the forward path from source to destination transferred all of its data to the subsequent node on the forward path as specified above. No two nodes along the forward path emitted data packets simultaneously. Additionally, data throughput from node to node was comparable to that achieved when transferring files using Compressed File Transfer Protocol (CFTP), a file transfer client defined within STANAG 5066.

Next, an acknowledgment packet was forwarded along the reverse path. Once the acknowledgment packet was received by the original source, HF-DSR displayed message transfer success.

Trails conducted indicate that HF-DSR is effective in networks that have linear characteristics. Networks such as this have relatively few possible routes from arbitrary sources and destinations.

6.1.1.4.1.2Diamond Networks

Path discovery proceeded as expected within diamond networks. Path requests were simultaneously transferred from N_0 to N_1 and N_2. Once the path request from N_0 was received, N_1 and N_2 submitted path request packets to their respective STANAG 5066 protocol stacks at the same time. This can cause three scenarios to occur. The first two scenarios are that either N_1 or N_2 establishes a connection to N_3. The third scenario is that neither N_1 nor N_2 successfully contact N_3. If this occurs, then path discovery fails. However, the first two possibilities are much more likely due to the radios' ability to resolve simultaneous ALE link requests.

If either N_1 or N_2 successfully established a connection with N_3 during path discovery, then the path request packet was forwarded to N_3. Upon receiving the path request, N_3 immediately submitted a reverse path request to its STANAG 5066 protocol stack. N_1 and N_2 received the reverse path request, processed it, and simultaneously submitted it to their respective STANAG 5066 protocol stacks. Similarly to the manner in which path request packets were simultaneously forwarded, N_1 and N_2 transmitted ALE link establishment requests to N_0 at the same time. In this manner, either N_1 or N_2 established a link with N_0. Next, the reverse path request was sent to N_0. At this point, path discovery was complete.

Data transfer operated successfully within diamond networks. There were no issues with data collisions that were experienced with path discovery. In all respects, data transfer functioned just as it did in linear networks, except that the forward and reverse paths were not necessarily symmetrical. This is due to the simultaneous ALE transmissions during path discovery being resolved differently per attempt.

Trials conducted indicate that HF-DSR requires effective collision avoidance mechanisms to overcome simultaneous transmissions. The collision avoidance mechanisms have to become more effective in to overcome corresponding increases in network connectivity.

Contrasting test results in the “Linear” network and the “Diamond” network suggests that increasing network connectivity increases the likelihood that multiple nodes will attempt to transmit simultaneously during path discovery. These simultaneous transmissions could potentially interfere with one another. However, testing within the diamond network showed that as long as one of the simultaneous transmissions is received, path discovery is relatively unaffected. As a result, HF-DSR requires underlying communications protocols that are resistant to interference.

7 Future Work

Incorporating additional DSR features into HF-DSR could reduce the network bandwidth required to conduct path discovery. Allowing an intermediate node N_i to transmit reverse path request packets on behalf of the intended destination N_d would shorten the path discovery process immensely. N_i would use a cached path between itself and N_d to fill in the reverse path request fields. This mechanism would reduce the number of path discovery hops between the same two nodes as long as the cached paths at intermediate nodes are not expired.

DSR references a mechanism called "passive acknowledgment" [Jubin+, 1987] which allows N_i to confirm data transfer with N_{i+1} by monitoring the data being transferred from N_{i+1} to N_{i+2}. This mechanism was originally proposed for implementation in HF-DSR. However, discussions between the author and project advisor allowed for this feature to be referenced in the "Future Work" section instead.

DSR references a "hop limit" that limits route lengths as well the quantity of network traffic utilized during path discovery. Such a feature would be ideal for HF radio networks, and could be included in future HF-DSR versions.

Additional testing with different and larger network topologies can be used to optimize HF-DSR for wider applications. In this manner, unanticipated scenarios will be encountered in a controlled environment. Mesh and star networks, which are common HF radio topologies, are suitable situations to analyze in such a manner. However, the testing could not be performed within the scope of this project due to resource limitations.

Obtaining data points while testing HF-DSR over-the-air would provide a "proof of concept" usable within papers and presentations. Although over-the-air test results are difficult to analyze and quantify beyond success and failure, they provide evidence that systems operate within certain parameters under certain conditions. Such test results would reduce doubt and increase confidence among the HF community with regard to HF-DSR. However, the testing could not be performed within the scope of this project due to resource limitations.

Insufficient data was taken during this project to analyze asymptotic running time bounds. Such an analysis could be the undergone by another project.

8 Conclusion

HF-DSR is a successful adaptation of DSR to the HF communication medium. HF-DSR test results are encouraging when using "linear" and "diamond" networks. Based on partial connectivity within the test networks exercised, HF-DSR should operate similarly over real-world networks that share the conditions tested.

HF-DSR was constructed as a client application to the NATO standard STANAG 5066, illustrating that HF-DSR capability can be added to any STANAG 5066 compliant radio system. HF-DSR uses standard S_PRIMITIVEs to communicate with STANAG 5066. As such, no implementation dependence is imposed on any system using HF-DSR.

Ad-hoc route discovery and file transfer on small HF radio networks was successful. As a proof of concept, this project demonstrates that ad-hoc routing and data transfer is feasible and valuable for HF radio networks.

9 References

[CDMA] CDG: Technology: Welcome to the World of CDMA. (1999). Retrieved May 24, 2004 from http://www.cdg.org/technology/cdma%5Ftechnology/a%5Fross/cdmarevolution.asp.

[Holcomb+, 2003] Holcomb, M.T. and Weston, J.H. Enhancing Channel Utilization and Performance of STANAG 5066. In Ninth International Conference on HF Radio Systems and Techniques. Institute of Electrical Engineers, London: 2003.

[IANA] Internet Assigned Numbers Authority. (2003, September 22). Retrieved February 2, 2004 , from http://www.iana.org/assignments/port-numbers

[Johnson+, 2001] David B. Johnson and David A. Maltz. DSR: The Dynamic Source Routing Protocol For Multihop Wireless Ad Hoc Networks. In Perkins, Charles. Ad Hoc Networking. Addison-Wesley, New York, 2001.

[Jubin+, 1987] J. Jubin and J.D. Tornow. The DARPA Packet Radio Network Protocols. Proceedings of the IEEE 75(1):21-32, January 1987.

[Clausen+, 2003] T. Clausen and P. Jacquet. RFC3626: Optimized Link State Routing Protocol (OLSR). Retrieved December 11, 2005 from http://ietf.org/rfc/rfc3626.txt

[STANAG 5066] NATO Standardization Agreement, "Profile for High Frequency (HF) Radio Data Communications STANAG 5066" Version 1.2

[TDMA] IEC: Time Division Multiple Access (TDMA). (2003). Retrieved May 24, 2004 from http://www.iec.org/online/tutorials/tdma/.