

# A Split Implementation of the Dynamic Source Routing Protocol for Lunar/Planetary Surface Communications

Jerry Toung  
Advanced Management Technology, Inc.  
NASA Ames Research Center  
M/S 258-6  
Moffett Field, CA 94035  
jtoung@arc.nasa.gov

Raymond Gilstrap, Kenneth Freeman  
NASA Ames Research Center  
M/S 258-5  
Moffett Field, CA 94035  
{ray.gilstrap, kenneth.freeman-1}@nasa.gov

*Abstract*—Future NASA exploration missions will involve teams of humans and robots working together to achieve science objectives on lunar and planetary surfaces. Members of these teams must be able to communicate with each other interactively as they work together in close proximity. However, current operational procedures and technologies are based on the assumption that surface elements operate in isolation and communicate solely with the Earth, either directly or through orbiting relays.

The use of direct wireless communications among local surface elements will be necessary to achieve optimal communications efficiency. However, the surface elements are mobile and may lose communication with one another, due to traveling either out of range or behind an obstruction. This problem can be addressed through the use of a mobile ad hoc network routing protocol, allowing nodes unable to communicate directly to remain in contact by relaying data through one or more intermediate nodes.

To test this method of dynamic surface-to-surface communications, we have implemented the Dynamic Source Routing (DSR) protocol in a UNIX-based test environment. DSR is an efficient routing protocol that allows independent wireless nodes to self-organize into an ad hoc network. To enhance performance, forwarding and routing functions are split between kernel and user space, respectively. We have conducted field testing to determine the performance and effectiveness of DSR in maintaining connectivity among mobile nodes in the presence of communications outages caused by distance or obstructions. The results suggest that mobile ad hoc routing is a promising basis for communications among surface elements.<sup>12</sup>

## TABLE OF CONTENTS

1. INTRODUCTION .....	1
2. RELATED WORK.....	2
3. DSR PROTOCOL OVERVIEW.....	2
4. DSR IMPLEMENTATION .....	3
5. FIELD TESTS .....	4
6. CONCLUSIONS.....	5
7. FUTURE WORK .....	6
APPENDIX A. DSRD ALGORITHM.....	6

APPENDIX B. KERNEL MODULE ALGORITHM .....	6
REFERENCES .....	7
BIOGRAPHY .....	7

## 1. INTRODUCTION

NASA surface exploration missions to date have featured humans and robots operating essentially independently. The communications requirements for these missions have been relatively simple — the surface elements needed simply to communicate back to Earth, either directly or via an orbiting relay satellite. In contrast, future NASA surface exploration missions will incorporate teams of humans and robots working together to achieve science and engineering goals on planetary surfaces. For example, a geologist collecting samples on the lunar surface may work with a robotic assistant to annotate or analyze those samples [9].

As a result, flexible and dynamic planetary communications are critical to the success of NASA’s space exploration vision. Flexible on-site surface-to-surface communications would enable the planetary in-situ human and robotic teams to collaboratively adjust their activities based on unfolding situations. However, the use of surface-to-surface communications would represent a fundamental shift in communications support for NASA space missions. During the lunar missions of the Apollo era, astronaut communications and directives were relayed back to Earth. As a result, astronaut exploration time was not well utilized.

Today, NASA is moving towards humans controlling robotic assets in-situ. However, based upon today’s operations models and technology, communications between surface elements would still be relayed via the Earth or orbiting platforms, thus introducing long delay. In the case of the Spirit and Opportunity rovers from the Mars Exploration Rover Mission [4], landed assets are communicating via both an orbiting platform, the Mars Global Surveyor [5], and NASA’s Deep Space Network (DSN) [3] ground terminals. Due to the large physical distances, there is long latency between the landed rovers and the Earth based communications assets, measured in minutes, as opposed to the milliseconds on typical Earth-based communications systems. Similarly, communications between the Moon and Earth are measured in seconds. Furthermore, communications assets must be scheduled, based upon orbital positions, antenna directions and ground

<sup>1</sup> U.S. Government work not protected by U.S. copyright.

<sup>2</sup> IEEEAC paper #1189, Version 4, Updated Dec. 12 2005

station availability. For instance, if the orbiting platform were out of the line of sight, communications would not be possible. In the case of future lunar missions, this could prevent timely communications between landed elements that are a few meters apart. NASA's traditional use of scheduled point-to-point downlink of mission data could be enhanced with the inclusion of a local dynamic routed wireless communications architecture on the planetary surface.

As a result, we began exploring mobile ad hoc routing protocols, based upon work of the Internet Engineering Task Force's (IETF) Mobile Ad-hoc Networks (MANET) Working Group. Due to the potential of low power and low bandwidth mobile nodes on the lunar surface, we studied reactive routing protocols, as opposed to the more traditional proactive protocols. Since reactive protocols initiate routing on an on-demand basis, as opposed to sending periodic routing table updates like proactive protocols, there is a reduction of routing load [11]. This is advantageous in constrained power and bandwidth environments.

## 2. RELATED WORK

Routing data efficiently in a mobile ad hoc network can be challenging, and several protocols have been developed to solve this problem. The MANET working group has explored a number of these, including Dynamic Source Routing (DSR) [1], Ad Hoc On-demand Distance Vector (AODV) [6], Dynamic MANET On-demand (DYMO) Routing [7], Dynamic Destination-Sequenced Distance-Vector (DSDV) Routing [8], Optimized Link State Routing (OLSR) [12], and Topology Broadcast based on Reverse-Path Forwarding (TBRPF) [13]. Of these protocols, MANET chose to focus on DSR, AODV, OLSR, and TBRPF. OLSR and TBRPF are proactive protocols and, as mentioned above, less desirable for planetary surface communications. Of the two reactive protocols, AODV and DSR, we ultimately chose to work with DSR because of its ability to maintain a routing table with multiple paths. Because of the unpredictability of node movements and obstructions in the area being explored, path redundancy is a critical requirement.

Many of the studies of these protocols are simulation based. Furthermore, these simulations focus on the typical Department of Defense problem, where there are a large number of mobile nodes moving rapidly. In the case of lunar exploration, there will likely be a small number of mobile nodes exploring the planetary surface. Also, the studies that do involve live tests generally happen in a university campus setting, with buildings and vehicles serving as the main obstructions. As a result, we chose to implement and validate DSR in an environment that more closely resembles actual lunar and planetary surface conditions.

## 3. DSR PROTOCOL OVERVIEW

This section provides an overview of the basic DSR operations; a more detailed discussion is available in the Internet Draft [1]. DSR is an efficient routing protocol that allows independent wireless mobile nodes to self-organize into an ad hoc network. The protocol specifies two main operations, *route discovery* and *route maintenance*, which allow nodes to learn and track routes to arbitrary destinations in the network.

### 3.1 Route Discovery

The operation of the protocol is illustrated in Figure 1. Time increases in the downward direction in the figure. The initial phase is route discovery, in which a node S wishing to send a packet to a destination node D broadcasts a *RouteRequest* (RREQ) message for D to the network. This message is contained in an IP packet that includes a DSR

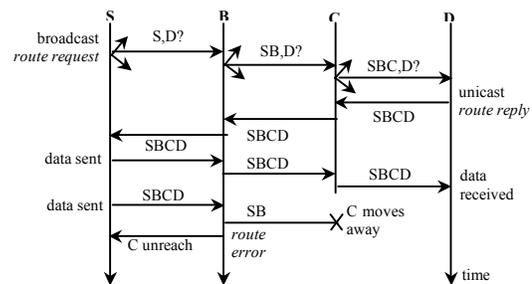


Figure 1 – DSR Protocol Operation

header preceding the transport protocol packet. The header includes the type of message, as well as the path taken by the packet so far. Initially, the path just contains S. This message propagates to the immediate neighbors of S, including B. In turn, each neighbor appends itself to the path recorded in the header and then propagates the RREQ to each of its own neighbors, such as C in the figure. This process repeats until the RREQ reaches D. D then issues a *RouteReply* (RREP) message, which travels back to S along the reverse of the recorded path. S then caches the route for future use, specifying the full route to D in subsequent data packets.

When S receives more than one RREP for a given destination, it chooses the first route that it receives in order to minimize the time for route discovery to take place. With minor modifications, the implementation can choose a route based on other metrics, such as previously observed throughput or packet loss rate for each node along the path.

### 3.2 Route Maintenance

Route maintenance is the mechanism by which S detects during transmission if its route to D has become invalid,

typically due to an intermediate node in the path failing or moving out of communication range. Path validity is monitored on a per-hop basis, with each node along the path using an acknowledgement mechanism to ensure that a packet was received by its downstream neighbor. This acknowledgement mechanism may be provided by the underlying layer 2 protocol (such as IEEE 802.11), or else a node may infer acknowledgement from overhearing its downstream neighbor relay the packet (for example, B concludes that C successfully received a packet after overhearing C transmit the packet to D). If neither of these mechanisms is available, DSR can rely on its own acknowledgement scheme, in which a node sends an *AcknowledgmentRequest* message to its downstream neighbor and awaits a corresponding *AcknowledgmentReply* message.

Regardless of the mechanism used, if a node does not receive an acknowledgement from a downstream neighbor, it assumes that neighbor is unreachable and marks invalid all routes in its cache that contain that neighbor. The node then issues a *RouteError* (RERR) message to all upstream nodes that have recently used the invalidated routes. The upstream nodes can then attempt to use other routes in their route caches, or they can invoke route discovery again to find new routes that do not include the failed node.

## 4. DSR IMPLEMENTATION

Implementing the dual operations of routing and forwarding in a mobile ad-hoc network routing protocol poses challenges in most operating systems. Packet forwarding refers to the process of sending a packet to the next hop on the path toward its destination, as determined by consulting a table (the forwarding table). Forwarding is implemented inside the kernel to maximize performance. Packet routing refers to the process of building the forwarding table, by communicating with neighboring nodes to learn enough of the network topology to determine the next hop to various destinations within the network. Routing is normally implemented in user space as daemon program, to avoid burdening the kernel with the overhead of communicating with other hosts and computing routes.

DSR and other on-demand routing protocols combine these functions, and so pose several implementation challenges [2]. One major issue is this intermixing of the forwarding and routing functions. Since these normally take place at different layers of the operating system, a choice of how to combine them is necessary. A complete in-kernel implementation would minimize expensive copying of packets between the kernel and user space, but would require heavy modifications to the IP stack and would impose the above communication and computation overhead on the kernel. A complete user-space approach is much simpler to implement, but forwarding performance will suffer because each packet would be copied into user

space for forwarding.

A second issue is the need for a mechanism to handle outstanding packets. Because routes which do not exist a priori need to be discovered before packets can be sent to the corresponding destination, the outstanding packets must be queued while route discovery takes place.

### 4.1 Implementation Approach

Kawada et al. [2] introduce a split kernel-user design, requiring minimal modifications to the kernel source. The idea was to keep the forwarding and routing functions in their natural domains, while enabling the communication between the two functions necessary for the operation of the protocol. Their Linux-based DSR was designed around a custom *Ad-hoc Support Library (ASL)*, consisting of a user-space *Routing Daemon* and two kernel modules: a *DSR-forwarding-helper* and a *DSR-maintenance-helper*. This design has the advantages of both optimal performance and simple implementation.

Because of the advantages of Kawada’s design, we decided to use it as a model for our UNIX-based implementation of DSR (Figure 2). We utilized FreeBSD, an operating system based on the Berkeley Software Distribution (BSD) version of UNIX, running on Intel-based hosts. Our implementation features a user-space daemon (*dsrd*) and a single kernel module (*if\_dsr.ko*), and requires minimal changes to the base FreeBSD code. The *dsrd* daemon performs the route request/maintenance functions, and uses system calls to populate the kernel-based forwarding table. The *if\_dsr.ko* module interacts with the IP stack to forward DSR packets.

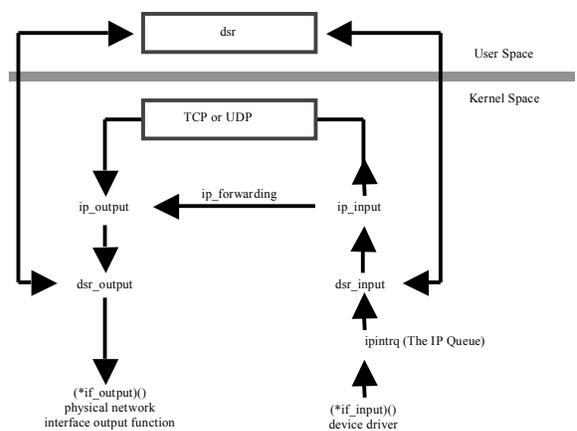


Figure 2 – DSR Node Structure

In the following sections, we will first present the system-specific implementation details. Also, we will present the algorithms of the Routing Daemon and the kernel module.

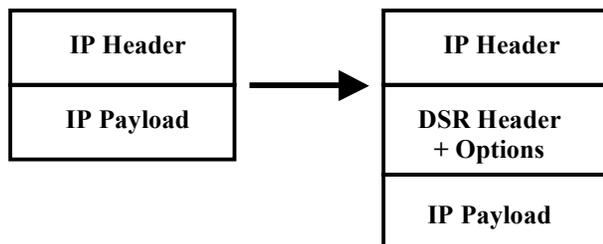
## 4.2 Implementation Details

The Dynamic Source Routing (DSR) protocol was implemented under FreeBSD 5.x and 6.0 using the dynamic kernel linker facility. Software developers utilize kernel modules, or KLDs, in order to implement new kernel functionality modularly, without needing to of reboot the system. Hence through KLDs, functionality can be dynamically added and removed while the system is running.

The DSR kernel module is implemented as a virtual network device driver and operates as follows:

*Incoming Packets* – The kernel module adds a new protocol switch input routine, *dsr\_input*, to the inbound IP stack. The existing IP module reads packets from the IP input queue (*ipintq*) as normal. For every packet processed by standard system call *ip\_input*, when the protocol field in the header indicates a DSR packet, the packet is passed to the *dsr\_input* function. *dsr\_input* processes DSR packets according to DSR specification [1]. After the routine is done with DSR-specific tasks, it passes the packet back to IP, which either transmits it to upper layer protocols like UDP or TCP (if the packet is destined for this host), or forwards it to the next hop in the network.

*Outgoing Packets* – The virtual interface *dsr0*, defined by *if\_dsr.ko*, accepts packets from the FreeBSD *ip\_output*



**Figure 3** – IP Packet Following DSR Header Insertion

function just like any other interface, but uses its own mechanism to arrange for their delivery via the actual physical interface. Packets are transferred to the DSR module on output by configuring the *dsr0* interface with an IP address in an administratively defined DSR subnet. This can be accomplished with a command such as:

```
ifconfig dsr0 10.10.1.1/24
```

This sets the IP address and subnet of *dsr0* and implicitly configures the host's routing tables such that any packet with a destination address in that subnet will be directed to *dsr0*. From there, *dsr\_output* will be called to further process packets sent to *dsr0* and forward them to the physical interface.

The algorithm for the *dsrd* routing daemon is listed in Appendix A. It handles the details of the route discovery and route maintenance operations, exchanging messages with both the kernel route cache and with corresponding routing daemons on other nodes.

Appendix B shows the kernel module algorithm. Three functions are of interest: *manet\_output*, *dsr\_output* and *dsr\_input*.

*manet\_output* receives messages sent by *dsrd*. These messages are placed in a FIFO queue called 'mnq'. Messages are transmitted between *dsrd* and the kernel module using a facility similar to a standard UNIX routing socket.

*dsr\_input* processes incoming IP packets that carry a DSR option header. The option header could either be a *RouteRequest*, *AcknowledgmentRequest*, *Acknowledgment*, or *SourceRoute* header. Once the pertinent information is retrieved from the packet, it is passed to the *dsrd* via the system function *raw\_input*.

*dsr\_output* receives packets from *ip\_output* and retrieves messages from the FIFO queue. It inserts a DSR option header in packets that don't already have one (Figure 3), or forwards packets that do.

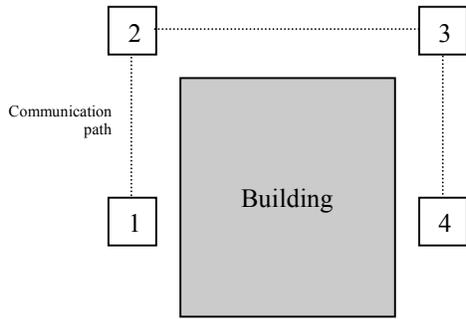
## 5. FIELD TESTS

The primary focus of this work was to determine the suitability of dynamically routed communications for surface mission environments. To that end, several sets of tests were performed to determine the effectiveness of DSR for enabling routing between hosts that cannot communicate directly. Throughput and latency measurements were also taken to determine the impact of DSR overhead. The tests used four laptops running DSR-enhanced FreeBSD, communicating via Lucent Orinoco 802.11b interfaces.

Three types of test environments were used:

*Firewall-based emulation* – The *ipfw* firewall, built into the FreeBSD kernel, was configured on each laptop to block traffic from one or two of the other laptops' MAC addresses. This allowed the laptops to remain in RF communication range of each other in a laboratory environment, while still providing communication outages between selected pairs of nodes.

*Building exterior* – The laptops were placed in locations on the outside of a building, as shown in Figure 4. This arrangement allowed each laptop to see its immediate neighbor(s), but the building blocked communication with other laptops.



**Figure 4** – Node Placement in Building-Exterior Tests

*Mars-like terrain* – The laptops were placed in rugged desert locations in Utah and Arizona, as well as a flat, rocky area in northern California. These environments mimic the type of terrain that might be encountered during an actual mission. In this case, communication outages may result from separation between nodes, obstructions, or in some cases destructive interference caused by multipath reflections. Figure 5 shows a representative arrangement.

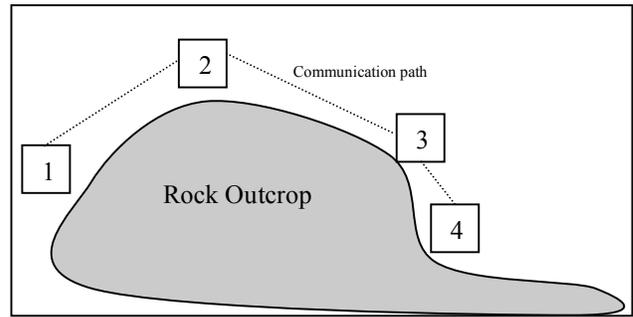
#### Test Results

The following results were observed:

*Route Discovery* – *dsrd* records a timestamped log of all routing messages and route cache updates. The standard ping utility was used to generate low-volume traffic to new destinations, and the *dsrd* logs and ping round trip times were observed. The log data indicated that the average elapsed time from the issue of an RREQ to the receipt of an RREP was approximately one second. However, as more hops were added between source and destination, the effects of transient packet losses between adjacent nodes during route discovery became more pronounced. For example, packets traveling from node 1 to node 4 can be affected by losses between any of the three pairs of adjacent nodes. In the worst case, successful route discovery in a four-node network took several tens of seconds. Further study is presently underway to characterize this behavior and correlate it with fluctuations in observed RF signal strength.

*Multihop routing* – DSR was able to successfully route packets between source and destination nodes separated by zero, one, or two intermediate nodes. In cases where the source or destination node was mobile, the route cache entry was updated when either the node lost contact with its next-hop neighbor or when the route expired.

*Latency/Jitter* – Between two nodes that are within communication range of each other, the observed latency using DSR was not measurably greater than the latency without DSR at the same separation distance. The latency measured during multiple-hop tests was more variable, since latency is affected by the completion time of the route



**Figure 5** – Node Placement in Mars-Like Terrain

discovery process as described above. When a valid route was already in the cache, round trip times were observed to be approximately equal to the sum of the propagation delays between adjacent nodes, or 30-40 milliseconds in a typical experiment. When route discovery was needed, the round trip times for the initial ping packets that triggered the discovery process were equal to the route discovery completion time.

*Throughput* – Throughput tests were performed using Iperf [10]. The tests were primarily done using the UDP transport protocol. This was done to gain a better understanding of the raw throughput of the protocol without the artifacts introduced by TCP’s response to the highly variable latency. Tests were run from node 1 to each of nodes 2, 3, and 4. Average throughput values are shown in Table 1. These values compare with typical observed throughput of 1 Mbps between adjacent nodes without DSR. As the table shows, DSR imposes a modest overhead on data transfers between adjacent nodes, while providing a useful data rate even to a node three hops away that is otherwise unreachable.

Path	Throughput (Kbps)
1-2	819
1-3	356
1-4	262

**Table 1** – Throughput Results

## 6. CONCLUSIONS

Our test results indicate that ad hoc routing protocols such as DSR have the potential to greatly increase the flexibility of surface communications. By allowing surface nodes to relay data among themselves, the effective communication range of a surface-based workgroup can easily adapt to nodes moving about a work area in the presence of obstructions. Our test results also suggest that more work is needed to make the route discovery process more robust in the presence of transient packet losses.

## 7. FUTURE WORK

Our next goal is to get further experience with ad hoc routing in realistic mission environments. To that end, we are planning to adapt our DSR implementation for use in a mission communications hardware testbed. This testbed will include a channel path simulator that can replicate the exact RF environment that will be encountered on the lunar or Martian surface.

### APPENDIX A. DSRD ALGORITHM

```
initialize RouteCache
initialize RouteRequestTable
initialize DaemonLogfile
m_sock = communication socket between dsrd and the kernel module
s_sock = raw IP socket
```

```
loopforever 1
{
  process RouteRequestTable
  if m_sock bit is set
  {
    loopforever 2
    {
      read 'msg' from m_sock
      switch(msg)
      Case AddRoute:
        /* the DSR module is passing us source route
        information it got from a packet it received */.
        Add that source route into RouteCache.
        Break (leave loopforever2).

      Case Got an Acknowledgment:
        if MaintHoldOffTime has elapsed and this Ack
        comes from a previous host we know OR if this is
        coming from a different host than in the last
        Ack Request we issued, send Ack Reply message to dsr
        kernel module via m_sock socket.
        Break.

      Case Got RouteRequest:
        store source route information into RouteCache.
        Register the newly received Route Request in the
        RouteRequestTable if it's not already in there.
        Search the RouteCache for a route to the target node.
        If a route is found, build a RouteReply packet and
        send it back on s_sock socket.
        If no route is found rebuild the Route Request packet,
        decrement TTL and send it on s_sock socket.

      Case Get Route:
        /* we received a message from the dsr module saying
        that it needs the route to a certain destination.*/
        Search the RouteCache for a route to that destination.
        If a route is found, send a Route message (RT) on
        m_sock socket.
        If no route is found, build a Route Request packet and
        send it out via the s_sock socket.

      Case Got RouteReply:
        if our IP address is present in the RouteReply source
        route, store in the RouteCache the path starting at
        our IP address and on.
```

Write this newly acquired source-route on m\_sock socket. It could be that the module needs it.

Case Got AcknowledgmentRequest:

build an IP packet containing a DSR header with Acknowledgment field set.  
Send that packet on s\_sock socket.

```
Case Got RouteError:
/*We receive 3 IP address from the module. (E) is the node
That detected that (U) has become unreachable and that
(S) needs to be notified */
Update RouteCache based on that information. Remove
any source route with broken link E -> U.
Search for a route in RouteCache to destination S.
If route is found
Build SourceRoute+RouteError packet and send it on
s_sock socket.
Else
Build RouteRequest packet to target node S and send it on
S_sock socket.
}
}
```

### APPENDIX B. KERNEL MODULE ALGORITHM

```
loopforever
{
  read 'mq' FIFO
  switch(msg.type)
  case RT:
    /* This message contains source route to a target node*/
    look at all the outstanding packets in the SendBuffer and
    send those that needs to go to that target by using this source
    route information.
    Put a copy of every packets sent in the MaintenanceBuffer.
    Break

  case ACKREP:
    /* we have received an acknowledgment from a host 1-hop away*/
    Remove every packet in the MaintenanceBuffer that match this
    acknowledgment information (i.e dest IP addresses are the same).
    Break
}

whileloop on MaintenanceBuffer
{
  if a packet has been held for more than MaintainHoldOffTime &&
  if it has been sent more than MaxMaintRexmt
  Issue RouteError.
  Else if a packet retransmit count < MaxMaintRexmt
  resend the packet out.
  Increment the packet's retransmit counter.
}

switch(IP protocol)
{
  case IPPROTO_DSR:
    switch(dsr option)
    {
      case SourceRoute(SRCRT):
        if our IP address is listed in the source route path
        forward packet.
        Put a copy of the packet in the MaintenanceBuffer.
      Else
        discard packet.
    }
    Break

  case Acknowledgment (ACK):
  case AcknowledgmentReply (ACKREQ):
    send packet out if we are the source.
    Break
}
```

```

    case RouteRequest (RREQ):
        set destination address to IP limited broadcast address =
        255.255.255.255
        send packet out.
        Break.
    }
J
case ICMP or TCP or UDP or IP:
    if we don't know the route to the destination,
        issue a GetRoute (GETRT) message to the RoutingDaemon
        Put packet in the SendBuffer.
    Else
        build a SourceRoute and AcknowledgmentRequest
        options in a DSR header.
        Insert the DSR header into the packet after the IP header.
        Put a copy of the packet on the MaintenanceBuffer
        Send packet out.

Break.
)

```

## REFERENCES

- [1] D.B. Johnson, D.A. Maltz, Y Hu, Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR), <http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt>, July 2004.
- [2] V. Kawadia, Y. Zhang, B. Gupta, System services for implementing ad-hoc routing protocols, Proceedings of the International Conference on Parallel Processing Workshops, 2002.
- [3] Deep Space Network (DSN), <http://deepspace.jpl.nasa.gov/dsn/>
- [4] Mars Exploration Rover Mission, <http://marsrovers.jpl.nasa.gov/home/index.html>
- [5] Mars Global Surveyor Mission, <http://marsprogram.jpl.nasa.gov/mgs/>
- [6] C. Perkins, E. Belding-Royer, S. Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. <http://www.ietf.org/rfc/rfc3561.txt>, July 2003.
- [7] I. Chakeres, E. Belding-Royer, Dynamic MANET On-demand (DYMO) Routing, <http://www.ietf.org/internet-drafts/draft-ietf-manet-dymo-02.txt>, June 2005.
- [8] C.E. Perkins, P. Bhagwat, Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers, In Proceedings of ACM SIG-COMM'94, Lonkon, U.K., September 1994.
- [9] W. Clancey. "Mobile Agents Project." <http://is.arc.nasa.gov/HCC/tasks/MblAgt.html>
- [10] Iperf Bandwidth Measurement Tool, <http://dast.nlanr.net/Projects/Iperf/>
- [11] C.E. Perkins, E.M. Royer, S.R. Das, M.K. Marina, Two On-Demand Routing Protocols for Ad Hoc Networks, IEEE Personal Communications, February 2001
- [12] T. Clausen, P. Jacquet. Optimized Link State Routing Protocol (OLSR). <http://www.ietf.org/rfc/rfc3626.txt>, October 2003.
- [13]. Topology-Based Reverse Path Forwarding. <http://tbrpf.erg.sri.com>

## BIOGRAPHY

Jerry Toung is a Network Software Engineer with Advanced Management Technology, Inc. (AMTI) at NASA Ames Research Center. He received a Diplome d'Ingenieurs (MSEE) in 2000 from Polytech Nantes in Nantes, France. He has been involved in the design, development and testing of networking software with the NREN group for over 5 years. In addition to DSR, his projects include a high-performance UNIX workstation-based network monitoring system and network traffic analysis tools.

Raymond Gilstrap is a network engineer with the NASA Research and Engineering Network (NREN) group at NASA Ames Research Center. He received a B.S. in Electrical Engineering at Florida Agricultural and Mechanical University in 1995, an M.S. in Electrical Engineering from the University of California, Berkeley in 1997. He has been involved in numerous projects since joining NASA in 1998, including development of the PCMon measurement and monitoring tool, performance engineering to support a real-time application over satellite between the U.S. and Japan, and providing engineering support for several scientific field experiments.

Kenneth Freeman has been manager of the NREN group at NASA Ames Research Center for the last five years. He received a B.S. in Electrical Engineering and Computer Science from the University of California, Berkeley in 1988 and an M.S. in Electrical Engineering from San Jose State University, San Jose, CA in 1990. He joined NASA in 1991 and was instrumental in the design of the original NREN testbed. He has played major roles in several engineering projects. Examples include development of the Next Generation Internet Exchange (NGIX) West; development of the first IP Multicast exchange point; development and deployment of QoS, measurement and monitoring, and IPv6 technologies on NASA networks; and prototyping of a wide range of high-performance networking applications, from remote echocardiography to air-traffic management.