Implemention of Service Discovery in Mobile Ad-hoc Networks in ns-2.26 (with AODV as routing protocol)

Sandeep Gupta
M.Tech-WCC, 2nd sem
Indian Institute of Information Technology, Allahabad

sandeepgupta@iiita.ac.in

May, 2003

Abstract

The following work is based on IETF's draft-koodli-manet-servicediscovery-00.txt draft (Appendix A) by Rajeev Koodli and Charles E. Perkins. Nodes in a manet can offer various services like printer, name service, smtp etc. Other nodes wishing to access these services send route request messages (RREQ) with service discovery extensions and get route reply (RREP) with the address of the node providing the service. This draft has been implemented in latest available version of ns, ie. 2.26 with gcc 3.2 compiler on SuSE Linux 8.1

1 Overview of Service Discovery

Nodes that are willing to access services like printer, name service, smtp etc. provided by other nodes must send a query asking for the same. This query can be based on either service name or port number. These parameters are called "service selectors". These need to be mapped to IP addresses. Additionally, routes to these resolved addresses need to be discovered. This process is termed as service discovery.

2 Service Discovery and Proactive protocols

Service discovery in proactive protocols like DSDV, OLSR is trivial. One needs to add service extensions to the topology update packets and information regarding services would be available to all nodes.

3 Service Discovery and Reactive protocols

Reactive protocols like AODV, DSR, do not send topology updates regularly but search for a route only when required. These protocols rely on RREQ and RREP messages for finding these routes. Service discovery, here, can take place by using extensions to these messages. A service query extension (SREQ) would contain either the name of the service or the port number. The destination address is usually zero (unknown/broadcast). A node offering this service or having a valid binding creates an RREP with service reply extension (SREP) and sends it back to the initiator.

4 Protocol Details

Two cases can be considered when a node wishes to discover a service:

- The node has neither a service binding nor a route to service provider : Here, the node creates an RREQ with service query extension (as shown in Appendix A). It sets the destination address and destination sequence number as unknown. It broadcasts this message. Any node receiving this message which does not have a valid service binding simply rebroadcasts the message. If it provides the service or has a valid route to the service provider, then it sends back RREP with SREP extension. If it has a service binding but no valid route to the service provider, then it sets the destination address as the resolved address and broadcasts the query. In the last case, it must send an SREP URL extension. This must be done to ensure that if an intermediate node has a valid route to the resolved address, it is able to formulate service reply.
- The node has a service binding but an expired route : This case is similar to the one discussed above. The node must send an SREP URL extension, for the reason discussed above.

Each service binding has an associated lifetime, after which it becomes invalid. Nodes constantly update their service tables if they get a "better" binding for a service. A "better" service is decided on the basis of number of hops to the provider and lifetime of the binding, in that order.

5 Overview of Working of the Implementation

Each node that wants to provide some service, binds itself to that service. Each service should have a port no. and service name associated with it. On the lines of routing table, there is a service table [aodv_rtable.h/cc] in which the service provider node adds its own binding with lifetime as infinite. All nodes use this service table for recording various service bindings offered in the network. This table has these fields: port no., service string, destination address (of service provider), expiry time of service binding, number of hops to the service provider.

Now, when a node wants to start service discovery, it must know either the service port number or the service name. Accordingly, it makes an RREQ message with service request (SREQ) port extension or SREQ URL extension. It also maintains a queue of service discoveries initiated by it. [aodv.h]

Each node receiving RREQ, first checks for SREQ extensions and processes it if they are present. If it has a service binding or it itself offers the service, then it can form a service reply extension (SREP) and send it back to the initiator of service discovery. If it has service binding but expired/no route to a service provider, it rebroadcasts the SREQ with destination set to that of the service provider. Otherwise, it simply rebroadcasts the request.

Service discovery initiator waits for a predetermined time (SDISEXTIME) before it reports failure.

Service predicate field in SREQ URL extension would depend on actual application sending the request. *eg.* application sending requests to find printer service can add that it needs laser printer, in predicate field.

Three commands, *viz.* 'svc-bind', 'svc-disc-port' and 'svc-disc-string', provide interface to tcl scripts for service binding, service discovery with port no. and service discovery with string/name, respectively

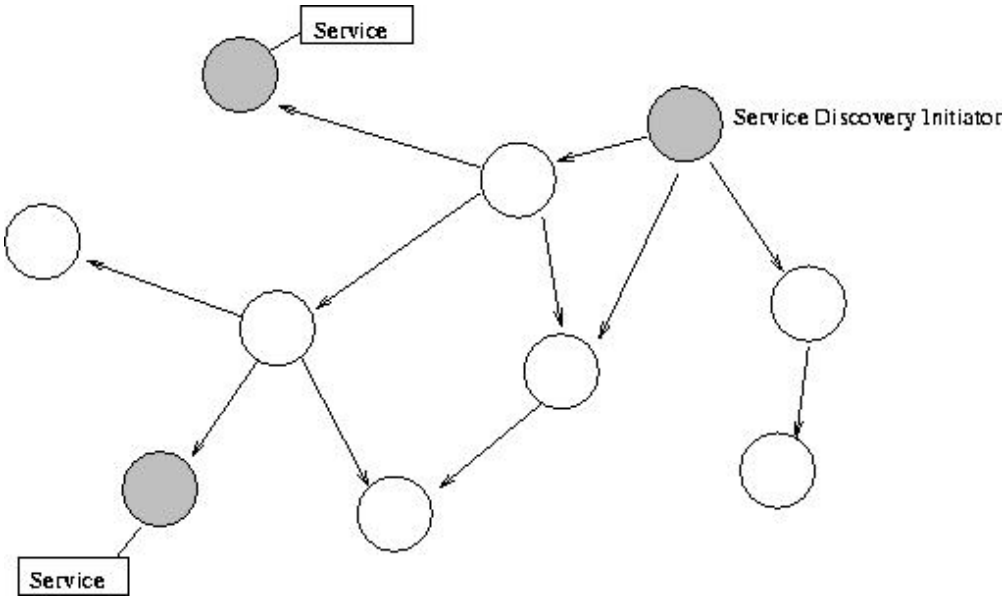

Figure 1: Service Discovery Request

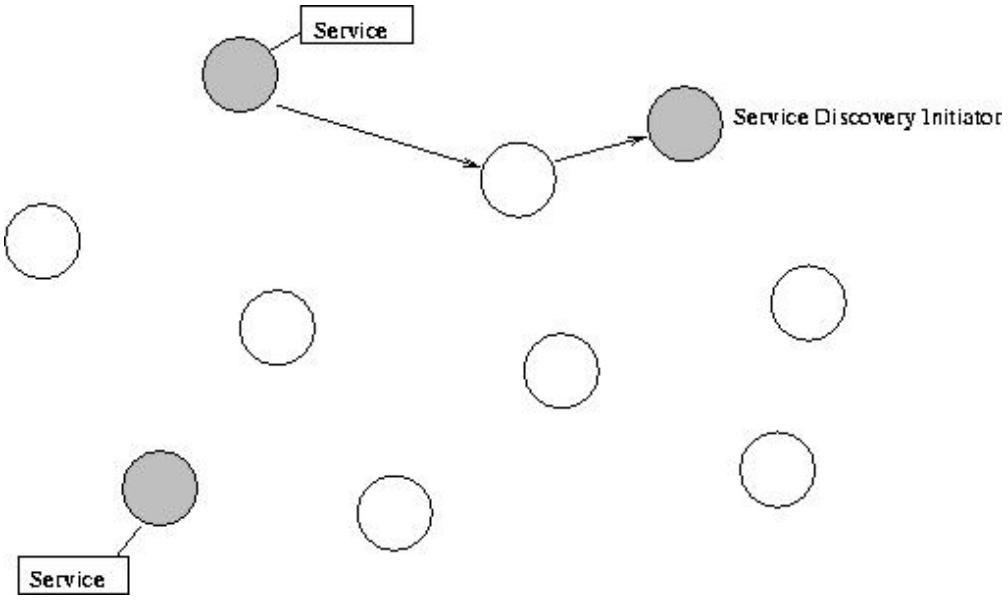

Figure 2: Service Discovery Reply

As is evident from figures 1 and 2, a node offering service which is nearer to the service discovery inititator is preferred.

Present implementation only prints out whether a service discovery was successful or not (and address of node providing service, in case of a success). Agents can easily be written on top of this implementation which can use it to find such service and then communicate with service providers.

6 Data Structures

6.1 Headers

6.1.1 Service Request Port Extension

```
struct hdr_aodv_sreq_port
{
  u_int8_t sq_type; //type
  u_int8_t sq_length; //length of header
  u_int16_t sqp_port; //port number
};
```

6.1.2 Service Request URL Extension

```
struct hdr_aodv_sreq_url
{
  u_int8_t sq_type;
  u_int8_t sq_length;
  u_int8_t squ_serv_length; //length of service type string
  u_int8_t reserverd;
  char squ_serv_string[32]; //service string
  char squ_serv_predicate[32]; //service predicate
};
```

6.1.3 Service Reply Extension

```
struct hdr_aodv_srep
{
  u_int8_t sp_type; //type
  u_int8_t sp_length; //length of header
  u_int16_t sp_lifetime; //lifetime of service/route

  u_int8_t sp_url_length; //length of url
  char sp_url[MAX_LENGTH_URL]; //url
  u_int8_t sp_no_url_auth; //# of autherization blocks
  char sp_auth_block[24]; //authorization block data
};
```

6.2 Service Discovery Table

```
//service table - based on routing table
//service table entry

class aodv_svc_entry
{
  friend class AODV;
  friend class aodv_svc_table;
public:
    aodv_svc_entry ();
protected:
    u_int16_t port;
  char serv_string[32];
  nsaddr_t dest;
  double expire;
    LIST_ENTRY (aodv_svc_entry) st_link;
};
```

```
//service table
class aodv_svc_table
{
  friend class aodv_svc_entry;
public:
    aodv_svc_table ()
  {
    LIST_INIT (\&sthead);
  }
  aodv_svc_entry *head ()
  {
    return sthead.lh_first;
  }
  aodv_svc_entry *st_add (u_int16_t port, char *serv_string, nsaddr_t dest,
                          double ltime);
  void st_delete (aodv_svc_entry * st);
  aodv_svc_entry *st_lookup (nsaddr_t id);
  aodv_svc_entry *st_lookup (u_int16_t port);
  aodv_svc_entry *st_lookup (char *serv_string);
  void st_purge (nsaddr_t selfid);
private:
  LIST_HEAD (aodv_sthead, aodv_svc_entry) sthead;
};
```

6.3 Service Discovery Queue (for keeping track of initiated discoveries)

```
struct servdiscoverQ
{
  u_int16_t port;
  char serv_string[32];
  double expire;
  servdiscoverQ *next;
};
```

The source code is presented in Appendix B.

7 Assumptions

Following assumptions have been made while implementing service discovery extension in AODV. A few of them modify the original draft.

1. Each service has an associated port number and a service string/name.
2. Every node that receives an SREQ (Service Request extension) or an SREP (Service Request extension) processes it. If it receives a reply and has not got that binding in its cache or it receives "better" binding, adds it to its cache. "Better" route is determined on the basis of number of hops to the service provider and lifetime of service binding, in that order.
3. Service predicate in SREP URL extension is only used to carry port number (if possible) and not other predicates like user name etc. (which would depend on actual application starting discovery)
4. Underlying routing protocol discovers and maintains routes.

5. If a node has a service binding but an expired route to the service provider, it broadcasts an SREQ URL extension, even if it got SREQ port extension. This is done so that an intermediate node that has a route to the service provider but no knowledge of service binding can also reply back with SREP.

6. Fields #URL auth. and URL auth. blocks in SREP are not used.

7. Default service binding time is 10 sec. This is the lifetime for which nodes receiving SREP will have a valid service binding.

8. Broadcast timer (which gets triggered every 6 sec) is used for purging out stale service bindings.

9. Each node waits for 5 sec after initiating service discovery. It reports a failure if it is not able to find any relevant service binding within this period.

10. Code type 129 (0x81), 130 (0x82) and 131 (0x83) have been assigned to SREQ Port extension, SREQ URL extension and SREP extension, respectively.

8 Suggested Modifications/Additions to the Draft

Following modifications/additions are suggested to the original draft :

1. Intermediate processing nodes receiving SREP should update their cache, if they do not have a valid service binding or they get a better service binding in SREP.

2. Whenever a node offering some service is about to go down or terminate its service, it should send an intimation to all active connections. It needs to send this to all active connections as it does not maintain a list of nodes accessing it service.

3. In case, when an intermediate node has a service binding but no valid route, it must make SREQ as URL extension, so that another processing node, that has valid route to the resolved address but no knowledge of service binding, can also reply back with SREP.

4. There should also be a predefined lifetime which the processing node sets in the above described case.

9 Acknowledgment

Thanks to Prof. Manish Singh, under whose guidance this project was successfully carried out.

10 Current Status

The work is available for download at www.iiita.ac.in/~sandeep_wc02. It includes installation instructions and a few sample simulation scripts. Source code can be modified without permission. The software/code is provided “as is” and the author is not liable for any damage/liability arising out of use of it.

References

- IETF's draft-koodli-manet-servicediscovery-00.txt, Rajeev Koodli and Charles E. Perkins, 2 October, 2002
- The ns manual (www.isi.edu/nsnam/ns)
- Ad-hoc Networking, edited by Charles E. Perkins, Addison - Wesley
- AODV's source code implementated in ns 2.26

Appendices

A IETF's Draft-koodli-manet-servicediscovery-00.txt

Manet Working Group
INTERNET DRAFT
2 October 2002

Rajeev Koodli
Charles E. Perkins
Communication Systems Laboratory
Nokia Research Center

Service Discovery in On-Demand Ad Hoc Networks
draft-koodli-manet-servicediscovery-00.txt

Status of This Memo

This document is a submission by the manet Working Group of the Internet Engineering Task Force (IETF). Comments should be submitted to the manet@ietf.org mailing list.

Distribution of this memo is unlimited.

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at:
http://www.ietf.org/ietf/1id-abstracts.txt
The list of Internet-Draft Shadow Directories can be accessed at:
http://www.ietf.org/shadow.html.

Abstract

In this document, we describe extensions to suitable ad hoc network routing protocols in order to provide support for service discovery. Typical Internet applications require information such as name resolution, a web proxy IP address, access to print services etc. While [manet] protocols have focussed on providing basic routing support for communication, this document addresses the problem of discovering services along with routes to those services.

Contents

1. Introduction

In this document, we describe extensions to suitable ad hoc network routing protocols in order to provide support for service discovery. Typical Internet applications require information such as name resolution, a web proxy IP address, access to print services etc. While [manet] protocols have focussed on providing basic routing support for communication, this document addresses the problem of discovering services along with routes to those services.

The problem of service discovery can be characterized as follows. Applications identify services using names or port numbers rather than IP addresses. These so-called "service selectors" need to be mapped to IP addresses. Routes need to be discovered to the resolved IP addresses, typically at the same time. The process of discovering the [service selector, IP address] binding is service discovery, whereas discovering a path to the resolved IP address is route discovery.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [1].

In addition, this document uses the following terms:

service selector
: Either a port number, or a Service URL optionally combined with some service attribute specification.

service binding
: An association between a service and the IP address of the node hosting the service application.

service request extension
: Either a Service Port Request extension (see section 7.1) or a Service URL Request extension (see section 7.2).

service request message
: A route discovery message containing a service request extension.

service reply message
: A route reply message containing a service reply extension.

Service URL
: A string as defined in RFC 2608 [2] useful for supplying information about network services.

3. Protocol Overview

4. Overview for Table-Driven Protocols

Service Discovery for table-driven prototocols such as OLSR or TBRPF can be managed by simply adding Service Reply extensions to the topology update routing protocol packets. In this way, service information can be made available immediately along with the information about which links are available for creating routes. In this case, the discovery operation is not separate from the operation of processing new topology update messages.

5. Overview for On-Demand Protocols

On-demand routing protocols for ad hoc networks typically offer a "Route Request" (RREQ) message to initiate the basic route discovery process, as specified in AODV [4] or DSR [3]. The basic service discovery process uses the same operations and message formats as route discovery, but with extensions with the format in either section 7.1 or 7.2. When a node needs to discover a service, it formulates a service request containing the service selector data which will identify the desired service application. The node then includes the query as an extension to the RREQ, and floods the RREQ. The service query requests resolution of service name into an IP address of a node that offers the requested service. By default, a service query also requests a route to the resolved IP address.

There are two kinds of extensions that can be used to request service. The Service Port Request extension (see section 7.1) includes the port number that is associated with the service application. The Service URL Request (see section 7.2) extension utilizes the data formats from Service Location Protocol [2] for more accurate service identification. These two extensions are called "service request" extensions.

A node that receives a RREQ with a service request extension (call such a message SREQ) has to perform the following. First, it must determine whether it has a valid service binding for the service selector present in the SREQ. Such a service binding would have a mapping of [service selector, IP address] with valid lifetime. Next, the receiving node must verify if there is a valid route available for the resolved IP address; that is, whether a valid

route (e.g., a source route, or a next hop) exists to the node that offers the service specified in SREQ. If these conditions are met, the processing node constructs a RREP and copies the received service selector data to a Service Reply extension to the RREP (call such a message SREP). Then the node transmits the SREP back towards the requester. The Service Reply extension supplies the requested binding with an associated Lifetime.

If the processing node has a service binding but no active route to the resolved IP address, it sets the Destination IP address to the resolved IP address, and re-broadcasts the RREQ message. Any node that receives a SREQ message with a non-zero Destination Address may send a SREP message if it has either a route to the destination, or a route to an equivalent service as described in the service request extension. If the extension is already present in a RREP packet, e.g., some other node (or the destination itself) provided the extension, and the service binding Lifetime is greater than that present in its local binding, the processing node forwards the RREP towards the source. If the extension is not present, or has a Lifetime shorter than that present in its local binding, then the processing node creates the SREP packet with the greater Lifetime and forwards it towards the source.

If a processing node does not have a service binding, it re-broadcasts the original SREQ packet.

6. Protocol Details for Reactive Protocols

We consider the following cases.

1. the source has neither a service binding nor an active route. This would be the case when a node is first attempting to discover a service and a route to that service.

2. the source has a service binding, but an expired route. This may happen due to mobility, link failures or other conditions typical in ad hoc networks.

3. the source has an active route but the service binding is either absent or has expired. The binding may not be present because the source may have communicated with the destination for reason(s) unrelated to the service in question. The binding may have expired because the source has not communicated with the destination for the reason of utilizing the service.

6.1. Service and Route Resolution

If a source node requires access to a service, and the underlying routing protocol supports on-demand route discovery, then the source creates a Service Request extension (using a message format defined in sections 7.1 or 7.2), include it in a RREQ message and broadcasts the resulting SREQ packet (as it does in typical on-demand routing protocols). The Destination Address MUST be set to zero, and the Destination Sequence Number (if required) MUST be treated as unknown.

An intermediate processing node treats the RREQ as a SREQ message since the service request extension is present. Whether or not the Destination Address is zero, the Service Request extension SHOULD be processed. If there is no service binding for the service name present in SREQ, the intermediate node MUST rebroadcast the packet. Depending on whether the intermediate has a valid route to the resolved IP address, there are two cases.

1. When the intermediate node has a valid route to the resolved IP address, it MUST create a Service Reply extension (whose format is defined in 7.3), include the extension in RREP and forward the resulting SREP packet to its previous hop.

2. When the intermediate node does not have a valid route to the resolved IP address, it MUST set the Destination Address to the resolved IP address and the Destination Sequence Number to the last known value in its route table for that destination, and rebroadcast the SREQ message.

6.2. Route Discovery only

The protocol behavior of the source is identical to the behavior of an intermediate node described in scenario 2 in Section 6.1 above.

6.3. Route is Present, Service Resolution Required

Since the source cannot determine that the IP address (for which the route is present) corresponds to the service name it is seeking to resolve, this scenario is same as in Section 6.1.

7. Extension Formats

The extensions introduced in this document use the Type-Length-Value (TLV) format, with 8-bit types.

7.1. Service Port Request

The extension format is shown below.

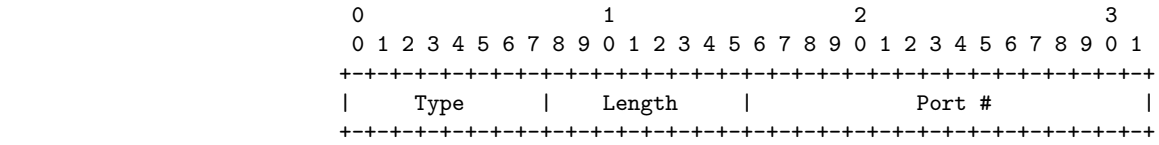

Figure 1: Service Port Request Extension Format

The Port # is the port number (for TCP or UDP) at which the service application is known to reside.

7.2. Service URL Request

The extension format is shown below.

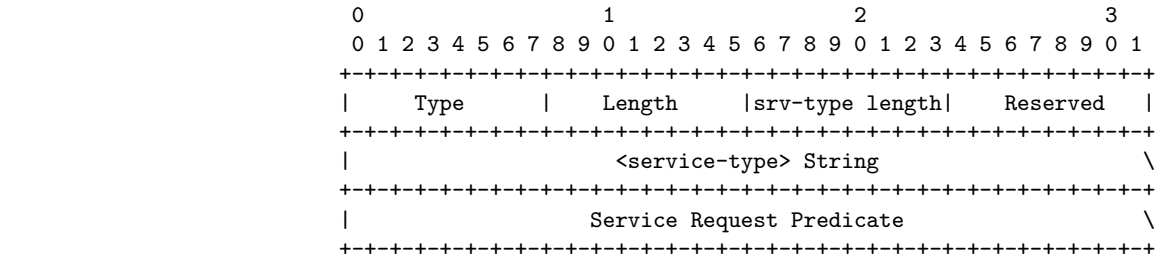

Figure 2: Service URL Request Extension Format

Type	TBD
Length	Length of the extension
srv-type> length	Length of the <service-type> string.
<service-type>	The <service-type> string [2].
Service Request Predicate	The <predicate> string [2].

The formats of the <service-type> and the "Service Request <predicate>" strings are defined by the Service Location Protocol [2].

7.3. Service Reply Extension

The extension format is shown below.

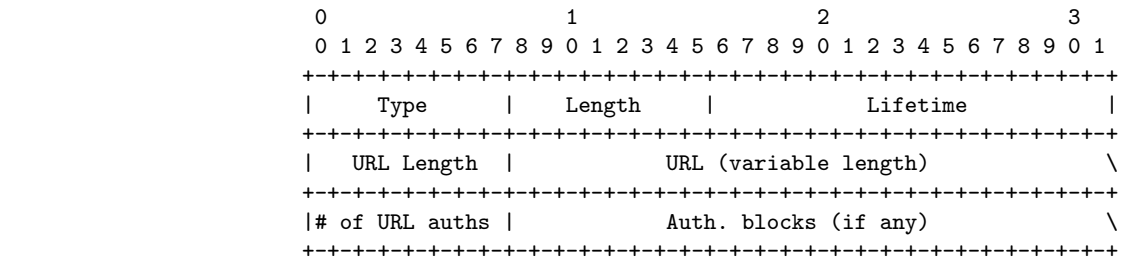

Figure 3: Service Reply Extension Format

The base format is shown in Figure 3. The header fields are described below. The format of the URL and the "Auth. block" strings are defined by the Service Location Protocol [2].

Type	TBD
Length	variable
Lifetime	The lifetime of the association between the service and the IP address of the node hosting the service.
URL Length	variable
URL	The service: URL strings as defined by the Service Location Protocol [2].
# of Auth. blocks	The number of Authorization blocks as defined by the Service Location Protocol [2].
Auth. block	The Authorization block data as defined by the Service Location Protocol [2].

8. Security Considerations

If the endpoints have a security association, the receivers of the SREQ and SREP messages can insert IPsec headers to assure that the senders are not masquerading using someone else's IP address. This does not, however, assure integrity of the routing path between sender and receiver. That would be a property of the underlying routing protocol. However, encryption could be employed to avoid unauthorized inspection of the data fields of the SREQ or SREP messages, regardless of the underlying routing protocol security.

The service authorization features from SLP have been maintained by including the Authorization Block data fields from the SLP Service Request message. No such feature is available when the port number alone is used in the service request message. Security may be provided by IPsec AH, if the endpoints have a suitable security association.

9. IANA Considerations

References

[1] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. Request for Comments (Best Current Practice) 2119, Internet Engineering Task Force, March 1997.

[2] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. Request for Comments (Proposed Standard) 2608, Internet Engineering Task Force, June 1999.

[3] D. Johnson and D. Maltz. Dynamic Source Routing in Ad-Hoc Wireless Networks. In Computer Communications Review -- Proceedings of SIGCOMM '96, August 1996.

[4] C. Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing (work in progress). Internet Draft, Internet Engineering Task Force. draft-ietf-manet-aodv-11.txt, July 2002.

Addresses

Questions about this memo can be directed to the authors:

Rajeev Koodli
Communications Systems Lab
Nokia Research Center
313 Fairchild Drive
Mountain View, CA 94043
USA
Phone: +1-650 625-2359
EMail: rajeev.koodli@nokia.com
Fax: +1 650 625-2502

Charles E. Perkins
Communications Systems Lab
Nokia Research Center
313 Fairchild Drive
Mountain View, CA 94043
USA
Phone: +1-650 625-2986
EMail: charliep@iprg.nokia.com
Fax: +1 650 625-2502

B Source Code

The source code files that were modified are presented here. Rest of the files, *viz.* aodv.tcl, aodv_logs.cc, aodv_rqueue.h and aodv_rqueue.cc remain unchanged. The complete source code is available at www.iiita.ac.in/~sandeep_wc02

B.1 aodv.h

```
/*Modified and extended by Sandeep Gupta, Indian Institute of Information
  Technology, Allahabad,
  India.
  May, 2003.
*/

/* Copyright (c) 2003 Indian Insitute of Information Technology, Allahabad.
  * All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted.
  *
  * The software is provided "as is" without express or implied warranty of any
  * kind arising out of the use of this software.
*/

/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice
and this permission notice appear in all copies of the software, derivative
works, or modified versions, and any portions thereof, and that both notices
appear in supporting documentation, and that credit is given to
Carnegie Mellon University in all publications reporting on direct
or indirect use of this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights to
redistribute these changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned
by Samir Das and Mahesh Marina, University of Cincinnati. The work was
partially done in Sun Microsystems.

*/

#ifndef __aodv_h__
#define __aodv_h__

//#include <agent.h>
//#include <packet.h>
//#include <sys/types.h>
//#include <cmu/list.h>
```

```
//#include <scheduler.h>

#include <cmu-trace.h>
#include <priqueue.h>
#include <aodv/aodv_rtable.h>
#include <aodv/aodv_rqueue.h>
#include <aodv/aodv_packet.h>

/*
  Allows local repair of routes
*/
#define AODV_LOCAL_REPAIR

/*
  Allows AODV to use link-layer (802.11) feedback in determining when
  links are up/down.
*/
#define AODV_LINK_LAYER_DETECTION

/*
  Causes AODV to apply a "smoothing" function to the link layer feedback
  that is generated by 802.11.  In essence, it requires that RT_MAX_ERROR
  errors occurs within a window of RT_MAX_ERROR_TIME before the link
  is considered bad.
*/
#define AODV_USE_LL_METRIC

/*
  Only applies if AODV_USE_LL_METRIC is defined.
  Causes AODV to apply omniscient knowledge to the feedback received
  from 802.11.  This may be flawed, because it does not account for
  congestion.
*/
//#define AODV_USE_GOD_FEEDBACK


class AODV;

#define MY_ROUTE_TIMEOUT        10 // 100 seconds
#define ACTIVE_ROUTE_TIMEOUT    10 // 50 seconds
#define REV_ROUTE_LIFE          6 // 5  seconds
#define BCAST_ID_SAVE           6 // 3 seconds


// No. of times to do network-wide search before timing out for
// MAX_RREQ_TIMEOUT sec.
#define RREQ_RETRIES            3
// timeout after doing network-wide search RREQ_RETRIES times
#define MAX_RREQ_TIMEOUT 10.0 //sec

/* Various constants used for the expanding ring search */
#define TTL_START     5
#define TTL_THRESHOLD 7
#define TTL_INCREMENT 2

// This should be somewhat related to arp timeout
#define NODE_TRAVERSAL_TIME     0.03 // 30 ms
#define LOCAL_REPAIR_WAIT_TIME  0.15 //sec

// Should be set by the user using best guess (conservative)
#define NETWORK_DIAMETER        30 // 30 hops

// Must be larger than the time difference between a node propagates a route
// request and gets the route reply back.

//#define RREP_WAIT_TIME     (3 * NODE_TRAVERSAL_TIME * NETWORK_DIAMETER) // ms
//#define RREP_WAIT_TIME     (2 * REV_ROUTE_LIFE)  // seconds
#define RREP_WAIT_TIME         1.0 // sec

#define ID_NOT_FOUND    0x00
#define ID_FOUND        0x01
//#define INFINITY        0xff

// The followings are used for the forward() function. Controls pacing.
#define DELAY 1.0 // random delay
#define NO_DELAY -1.0 // no delay
```

```
// think it should be 30 ms
#define ARP_DELAY 0.01 // fixed delay to keep arp happy


#define HELLO_INTERVAL          1 // 1000 ms
#define ALLOWED_HELLO_LOSS      3 // packets
#define BAD_LINK_LIFETIME       3 // 3000 ms
#define MaxHelloInterval        (1.25 * HELLO_INTERVAL)
#define MinHelloInterval        (0.75 * HELLO_INTERVAL)

//sandy
#define SDISEXTIME 5 //service discovery expire time 5sec
#define SBINDLTIME 10 //service binding lifetime 10sec

/*
  Timers (Broadcast ID, Hello, Neighbor Cache, Route Cache)
*/
class BroadcastTimer:public Handler
{
public:
  BroadcastTimer (AODV * a):agent (a)
  {
  }
  void handle (Event *);
private:
  AODV * agent;
  Event intr;
};

class HelloTimer:public Handler
{
public:
  HelloTimer (AODV * a):agent (a)
  {
  }
  void handle (Event *);
private:
  AODV * agent;
  Event intr;
};

class NeighborTimer:public Handler
{
public:
  NeighborTimer (AODV * a):agent (a)
  {
  }
  void handle (Event *);
private:
  AODV * agent;
  Event intr;
};

class RouteCacheTimer:public Handler
{
public:
  RouteCacheTimer (AODV * a):agent (a)
  {
  }
  void handle (Event *);
private:
  AODV * agent;
  Event intr;
};

class LocalRepairTimer:public Handler
{
public:
  LocalRepairTimer (AODV * a):agent (a)
  {
  }
  void handle (Event *);
private:
  AODV * agent;
  Event intr;
};
```

```
/*
  Broadcast ID Cache
*/
class BroadcastID
{
  friend class AODV;
public:
    BroadcastID (nsaddr_t i, u_int32_t b)
  {
    src = i;
    id = b;
  }
protected:
    LIST_ENTRY (BroadcastID) link;
  nsaddr_t src;
  u_int32_t id;
  double expire; // now + BCAST_ID_SAVE s
};

LIST_HEAD (aodv_bcache, BroadcastID);


/*
  The Routing Agent
*/
class AODV:public Agent
{

  /*
   * make some friends first
   */

  friend class aodv_rt_entry;
  friend class aodv_svc_entry; //sandy
  friend class BroadcastTimer;
  friend class HelloTimer;
  friend class NeighborTimer;
  friend class RouteCacheTimer;
  friend class LocalRepairTimer;

public:
    AODV (nsaddr_t id);

  void recv (Packet * p, Handler *);

protected:
  struct servdiscoverQ
  {
    u_int16_t port;
    char serv_string[32];
    double expire;
    servdiscoverQ *next;
  };
  servdiscoverQ *sdqhead , *sdqtail;
  int command (int, const char *const *);
  int initialized ()
  {
    return 1 && target_;
  }

  /*
   * Route Table Management
   */
  void rt_resolve (Packet * p);
  void rt_update (aodv_rt_entry * rt, u_int32_t seqnum,
  u_int16_t metric, nsaddr_t nexthop, double expire_time);
  void rt_down (aodv_rt_entry * rt);
  void local_rt_repair (aodv_rt_entry * rt, Packet * p);
public:
  void rt_ll_failed (Packet * p);
  void handle_link_failure (nsaddr_t id);
protected:
  void rt_purge (void);

  void enque (aodv_rt_entry * rt, Packet * p);
  Packet *deque (aodv_rt_entry * rt);
```

```
/*
 * Neighbor Management
 */
void nb_insert (nsaddr_t id);
AODV_Neighbor *nb_lookup (nsaddr_t id);
void nb_delete (nsaddr_t id);
void nb_purge (void);

/*
 * Broadcast ID Management
 */

void id_insert (nsaddr_t id, u_int32_t bid);
bool id_lookup (nsaddr_t id, u_int32_t bid);
void id_purge (void);

/*
 * Packet TX Routines
 */
void forward (aodv_rt_entry * rt, Packet * p, double delay);
void sendHello (void);
void sendRequest (nsaddr_t dst);
void sendSvcRequest (nsaddr_t dst, hdr_aodv_sreq sreq); //sandy

void sendReply (nsaddr_t ipdst, u_int32_t hop_count,
nsaddr_t rpdst, u_int32_t rpseq,
u_int32_t lifetime, double timestamp);
void sendSvcReply (nsaddr_t ipdst, u_int32_t hop_count,//sandy
   nsaddr_t rpdst, u_int32_t rpseq,
   u_int32_t lifetime, double timestamp,
   hdr_aodv_srep srep);
void sendError (Packet * p, bool jitter = true);

//sandy
void svc_discover (u_int16_t port, char serv_string[32]);
void svc_announce (u_int16_t port, char serv_string[32], nsaddr_t id);
void extract_serv_param (char *url, char *serv_string, u_int16_t * port);
void svc_purge ();

/*
 * Packet RX Routines
 */
void recvAODV (Packet * p);
void recvHello (Packet * p);
void recvRequest (Packet * p);
void recvReply (Packet * p);
void recvError (Packet * p);

/*
 * History management
 */

double PerHopTime (aodv_rt_entry * rt);


nsaddr_t index; // IP Address of this node
u_int32_t seqno; // Sequence Number
int bid; // Broadcast ID

aodv_rtable rthead; // routing table
aodv_ncache nbhead; // Neighbor Cache
aodv_bcache bihead; // Broadcast ID Cache

/*
 * Timers
 */
BroadcastTimer btimer;
HelloTimer htimer;
NeighborTimer ntimer;
RouteCacheTimer rtimer;
LocalRepairTimer lrtimer;

/*
 * Routing Table
 */
aodv_rtable rtable;
```

```
  aodv_svc_table stable; //sandy
  /*
   *  A "drop-front" queue used by the routing layer to buffer
   *  packets to which it does not have a route.
   */
  aodv_rqueue rqueue;

  /*
   * A mechanism for logging the contents of the routing
   * table.
   */
  Trace *logtarget;

  /*
   * A pointer to the network interface queue that sits
   * between the "classifier" and the "link layer".
   */
  PriQueue *ifqueue;

  /*
   * Logging stuff
   */
  void log_link_del (nsaddr_t dst);
  void log_link_broke (Packet * p);
  void log_link_kept (nsaddr_t dst);
};

#endif /* __aodv_h__ */
```

B.2 aodv.cc

```
/*Modified and extended by Sandeep Gupta, Indian Institute of Information
  Technology, Allahabad,
  India.
  May, 2003.
*/

/* Copyright (c) 2003 Indian Insitute of Information Technology, Allahabad.
  * All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted.
  *
  * The software is provided "as is" without express or implied warranty of any
  * kind arising out of the use of this software.
*/

/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice and this
permission notice appear in all copies of the software, derivative works, or
modified versions, and any portions thereof, and that both notices appear in
supporting documentation, and that credit is given to Carnegie Mellon University
in all publications reporting on direct or indirect use of this code or its
derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights to
redistribute these changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned by
Samir Das and Mahesh Marina, University of Cincinnati. The work was partially
done in Sun Microsystems. Modified for gratuitous replies by Anant Utgikar,
09/16/02.

*/

//#include <ip.h>

#include <aodv/aodv.h>
#include <aodv/aodv_packet.h>
#include <random.h>
#include <cmu-trace.h>
//#include <energy-model.h>

#define max(a,b)        ( (a) > (b) ? (a) : (b) )
#define CURRENT_TIME    Scheduler::instance().clock()

//#define DEBUG
//#define ERROR

#ifdef DEBUG
static int extra_route_reply = 0;
static int limit_route_request = 0;
static int route_request = 0;
```

```
#endif

//sandy

char *itoa (int);

/*
  TCL Hooks
*/


int
  hdr_aodv::offset_;
static class AODVHeaderClass:public PacketHeaderClass
{
public:
  AODVHeaderClass ():PacketHeaderClass ("PacketHeader/AODV",
sizeof (hdr_all_aodv))
  {
    bind_offset (&hdr_aodv::offset_);
  }
}
class_rtProtoAODV_hdr;

static class AODVclass:public TclClass
{
public:
  AODVclass ():TclClass ("Agent/AODV")
  {
  }
  TclObject *create (int argc, const char *const *argv)
  {
    assert (argc == 5);
    //return (new AODV((nsaddr_t) atoi(argv[4])));
    return (new AODV ((nsaddr_t) Address::instance ().str2addr (argv[4])));
  }
}

class_rtProtoAODV;


int
AODV::command (int argc, const char *const *argv)
{
  Tcl & tcl = Tcl::instance ();
  aodv_rt_entry *rt = rtable.rt_lookup (-1);
  if (!rt)
    rtable.rt_add (-1);
  if (argc == 2)
    {
      if (strncasecmp (argv[1], "id", 2) == 0)
{
  tcl.resultf ("%d", index);
  return TCL_OK;
}

      if (strncasecmp (argv[1], "start", 2) == 0)
{
  btimer.handle ((Event *) 0);

#ifndef AODV_LINK_LAYER_DETECTION
  htimer.handle ((Event *) 0);
  ntimer.handle ((Event *) 0);
#endif // LINK LAYER DETECTION

  rtimer.handle ((Event *) 0);
  return TCL_OK;
}
    }
  else if (argc == 3)
    {
      if (strcmp (argv[1], "index") == 0)
{
  index = atoi (argv[2]);
  return TCL_OK;
}
```

```
      else if (strcmp (argv[1], "log-target") == 0
       || strcmp (argv[1], "tracetarget") == 0)
{
  logtarget = (Trace *) TclObject::lookup (argv[2]);
  if (logtarget == 0)
    return TCL_ERROR;
  return TCL_OK;
}
      else if (strcmp (argv[1], "drop-target") == 0)
{
  int stat = rqueue.command (argc, argv);
  if (stat != TCL_OK)
    return stat;
  return Agent::command (argc, argv);
}
      else if (strcmp (argv[1], "if-queue") == 0)
{
  ifqueue = (PriQueue *) TclObject::lookup (argv[2]);

  if (ifqueue == 0)
    return TCL_ERROR;
  return TCL_OK;
}
      else if (strcmp (argv[1], "svc-disc-port") == 0)
{
  tcl.evalf ("puts \"%f\t%d: started service discovery at port %d\"",
     CURRENT_TIME, index, atoi (argv[2]));
  svc_discover ((u_int16_t) atoi (argv[2]), "");
  return TCL_OK;
}
      else if (strcmp (argv[1], "svc-disc-string") == 0)
{
  tcl.evalf ("puts \"%f\t%d: started service discovery for %s\"", CURRENT_TIME,
index,
     argv[2]);
  svc_discover (0, (char *) argv[2]);
  return TCL_OK;
}
    }
  else if (argc == 4)
    {
      if (strcmp (argv[1], "svc-bind") == 0)
{
  if (atoi (argv[2]) <= 0)
    {
      tcl.evalf ("puts \"%d: Invalid port no.\"", index);
      return TCL_OK;
    }
  stable.st_add ((u_int16_t) atoi (argv[2]), (char *) argv[3], index,
 INFLTIME,0);
  return TCL_OK;
}
    }
  return Agent::command (argc, argv);
}

/*
   Constructor
*/

AODV::AODV (nsaddr_t id):Agent (PT_AODV),
btimer (this), htimer (this), ntimer (this),
rtimer (this), lrtimer (this), rqueue ()
{


  index = id;
  seqno = 2;
  bid = 1;

  LIST_INIT (&nbhead);
  LIST_INIT (&bihead);

  logtarget = 0;
  ifqueue = 0;
  sdqhead=sdqtail=NULL; //sandy
}
```

```
/*
  Timers
*/

void
BroadcastTimer::handle (Event *)
{
  agent->id_purge ();
  agent->svc_purge ();
  Scheduler::instance ().schedule (this, &intr, BCAST_ID_SAVE);
}

void
HelloTimer::handle (Event *)
{
  agent->sendHello ();
  double interval = MinHelloInterval +
    ((MaxHelloInterval - MinHelloInterval) * Random::uniform ());
  assert (interval >= 0);
  Scheduler::instance ().schedule (this, &intr, interval);
}

void
NeighborTimer::handle (Event *)
{
  agent->nb_purge ();
  Scheduler::instance ().schedule (this, &intr, HELLO_INTERVAL);
}

void
RouteCacheTimer::handle (Event *)
{
  agent->rt_purge ();
#define FREQUENCY 0.5 // sec
  Scheduler::instance ().schedule (this, &intr, FREQUENCY);
}

void
LocalRepairTimer::handle (Event * p)
{ // SRD: 5/4/99
  aodv_rt_entry *rt;
  struct hdr_ip *ih = HDR_IP ((Packet *) p);

  /* you get here after the timeout in a local repair attempt */
  /*   fprintf(stderr, "%s\n", __FUNCTION__); */


  rt = agent->rtable.rt_lookup (ih->daddr ());

  if (rt && rt->rt_flags != RTF_UP)
    {
      // route is yet to be repaired
      // I will be conservative and bring down the route
      // and send route errors upstream.
      /* The following assert fails, not sure why */
      /* assert (rt->rt_flags == RTF_IN_REPAIR); */

      //rt->rt_seqno++;
      agent->rt_down (rt);
      // send RERR
#ifdef DEBUG
      fprintf (stderr, "Node %d: Dst - %d, failed local repair\n", index,
       rt->rt_dst);
#endif
    }
  Packet::free ((Packet *) p);
}


/*
   Broadcast ID Management  Functions
*/


void
AODV::id_insert (nsaddr_t id, u_int32_t bid)
```

```
{
  BroadcastID *b = new BroadcastID (id, bid);

  assert (b);
  b->expire = CURRENT_TIME + BCAST_ID_SAVE;
  LIST_INSERT_HEAD (&bihead, b, link);
}

/* SRD */
bool
AODV::id_lookup (nsaddr_t id, u_int32_t bid)
{
  BroadcastID *b = bihead.lh_first;

  // Search the list for a match of source and bid
  for (; b; b = b->link.le_next)
    {
      if ((b->src == id) && (b->id == bid))
return true;
    }
  return false;
}

void
AODV::id_purge ()
{
  BroadcastID *b = bihead.lh_first;
  BroadcastID *bn;
  double now = CURRENT_TIME;

  for (; b; b = bn)
    {
      bn = b->link.le_next;
      if (b->expire <= now)
{
  LIST_REMOVE (b, link);
  delete b;
}
    }
}

/*
  Helper Functions
*/

double
AODV::PerHopTime (aodv_rt_entry * rt)
{
  int num_non_zero = 0, i;
  double total_latency = 0.0;

  if (!rt)
    return ((double) NODE_TRAVERSAL_TIME);

  for (i = 0; i < MAX_HISTORY; i++)
    {
      if (rt->rt_disc_latency[i] > 0.0)
{
  num_non_zero++;
  total_latency += rt->rt_disc_latency[i];
}
    }
  if (num_non_zero > 0)
    return (total_latency / (double) num_non_zero);
  else
    return ((double) NODE_TRAVERSAL_TIME);

}

/*
  Link Failure Management Functions
*/

static void
aodv_rt_failed_callback (Packet * p, void *arg)
{
  ((AODV *) arg)->rt_ll_failed (p);
```

```
}

/*
 * This routine is invoked when the link-layer reports a route failed.
 */
void
AODV::rt_ll_failed (Packet * p)
{
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  aodv_rt_entry *rt;
  nsaddr_t broken_nbr = ch->next_hop_;

#ifndef AODV_LINK_LAYER_DETECTION
  drop (p, DROP_RTR_MAC_CALLBACK);
#else

  /*
   * Non-data packets and Broadcast Packets can be dropped.
   */
  if (!DATA_PACKET (ch->ptype ()) || (u_int32_t) ih->daddr () == IP_BROADCAST)
    {
      drop (p, DROP_RTR_MAC_CALLBACK);
      return;
    }
  log_link_broke (p);
  if ((rt = rtable.rt_lookup (ih->daddr ())) == 0)
    {
      drop (p, DROP_RTR_MAC_CALLBACK);
      return;
    }
  log_link_del (ch->next_hop_);

#ifdef AODV_LOCAL_REPAIR
  /* if the broken link is closer to the dest than source,
     attempt a local repair. Otherwise, bring down the route. */


  if (ch->num_forwards () > rt->rt_hops)
    {
      local_rt_repair (rt, p); // local repair
      // retrieve all the packets in the ifq using this link,
      // queue the packets for which local repair is done,
      return;
    }
  else
#endif // LOCAL REPAIR

    {
      drop (p, DROP_RTR_MAC_CALLBACK);
      // Do the same thing for other packets in the interface queue using the
      // broken link -Mahesh
      while ((p = ifqueue->filter (broken_nbr)))
{
  drop (p, DROP_RTR_MAC_CALLBACK);
}
      nb_delete (broken_nbr);
    }

#endif // LINK LAYER DETECTION
}

void
AODV::handle_link_failure (nsaddr_t id)
{
  aodv_rt_entry *rt, *rtn;
  Packet *rerr = Packet::alloc ();
  struct hdr_aodv_error *re = HDR_AODV_ERROR (rerr);

  re->DestCount = 0;
  for (rt = rtable.head (); rt; rt = rtn)
    { // for each rt entry
      rtn = rt->rt_link.le_next;
      if ((rt->rt_hops != INFINITY2) && (rt->rt_nexthop == id))
{
  assert (rt->rt_flags == RTF_UP);
  assert ((rt->rt_seqno % 2) == 0);
```

```
  rt->rt_seqno++;
  re->unreachable_dst[re->DestCount] = rt->rt_dst;
  re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
#ifdef DEBUG
  fprintf (stderr, "%s(%f): %d\t(%d\t%u\t%d)\n", __FUNCTION__,
   CURRENT_TIME, index, re->unreachable_dst[re->DestCount],
   re->unreachable_dst_seqno[re->DestCount], rt->rt_nexthop);
#endif // DEBUG
  re->DestCount += 1;
  rt_down (rt);
}
      // remove the lost neighbor from all the precursor lists
      rt->pc_delete (id);
    }

  if (re->DestCount > 0)
    {
#ifdef DEBUG
      fprintf (stderr, "%s(%f): %d\tsending RERR...\n", __FUNCTION__,
       CURRENT_TIME, index);
#endif // DEBUG
      sendError (rerr, false);
    }
  else
    {
      Packet::free (rerr);
    }
}

void
AODV::local_rt_repair (aodv_rt_entry * rt, Packet * p)
{
#ifdef DEBUG
  fprintf (stderr, "%s: Dst - %d\n", __FUNCTION__, rt->rt_dst);
#endif
  // Buffer the packet
  rqueue.enque (p);

  // mark the route as under repair
  rt->rt_flags = RTF_IN_REPAIR;

  sendRequest (rt->rt_dst);

  // set up a timer interrupt
  Scheduler::instance ().schedule (&lrtimer, p->copy (), rt->rt_req_timeout);
}

void
AODV::rt_update (aodv_rt_entry * rt, u_int32_t seqnum, u_int16_t metric,
 nsaddr_t nexthop, double expire_time)
{

  rt->rt_seqno = seqnum;
  rt->rt_hops = metric;
  rt->rt_flags = RTF_UP;
  rt->rt_nexthop = nexthop;
  rt->rt_expire = expire_time;
}

void
AODV::rt_down (aodv_rt_entry * rt)
{
  /*
   *  Make sure that you don't "down" a route more than once.
   */

  if (rt->rt_flags == RTF_DOWN)
    {
      return;
    }

  // assert (rt->rt_seqno%2); // is the seqno odd?
  rt->rt_last_hop_count = rt->rt_hops;
  rt->rt_hops = INFINITY2;
  rt->rt_flags = RTF_DOWN;
  rt->rt_nexthop = 0;
  rt->rt_expire = 0;
```

```
} /* rt_down function */

/*
  Route Handling Functions
*/

void
AODV::rt_resolve (Packet * p)
{
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  aodv_rt_entry *rt;

  /*
   *  Set the transmit failure callback.  That
   *  won't change.
   */
  ch->xmit_failure_ = aodv_rt_failed_callback;
  ch->xmit_failure_data_ = (void *) this;
  rt = rtable.rt_lookup (ih->daddr ());
  if (rt == 0)
    {
      rt = rtable.rt_add (ih->daddr ());
    }

  /*
   * If the route is up, forward the packet
   */

  if (rt->rt_flags == RTF_UP)
    {
      assert (rt->rt_hops != INFINITY2);
      forward (rt, p, NO_DELAY);
    }
  /*
   *  if I am the source of the packet, then do a Route Request.
   */
  else if (ih->saddr () == index)
    {
      rqueue.enque (p);
      sendRequest (rt->rt_dst);
    }
  /*
   *     A local repair is in progress. Buffer the packet.
   */
  else if (rt->rt_flags == RTF_IN_REPAIR)
    {
      rqueue.enque (p);
    }

  /*
   * I am trying to forward a packet for someone else to which
   * I don't have a route.
   */
  else
    {
      Packet *rerr = Packet::alloc ();
      struct hdr_aodv_error *re = HDR_AODV_ERROR (rerr);
      /*
       * For now, drop the packet and send error upstream.
       * Now the route errors are broadcast to upstream
       * neighbors - Mahesh 09/11/99
       */

      assert (rt->rt_flags == RTF_DOWN);
      re->DestCount = 0;
      re->unreachable_dst[re->DestCount] = rt->rt_dst;
      re->unreachable_dst_seqno[re->DestCount] = rt->rt_seqno;
      re->DestCount += 1;
#ifdef DEBUG
      fprintf (stderr, "%s: sending RERR...\n", __FUNCTION__);
#endif
      sendError (rerr, false);

      drop (p, DROP_RTR_NO_ROUTE);
    }
```

```
}

void
AODV::rt_purge ()
{
  aodv_rt_entry *rt, *rtn;
  double now = CURRENT_TIME;
  double delay = 0.0;
  Packet *p;

  for (rt = rtable.head (); rt; rt = rtn)
    { // for each rt entry
      rtn = rt->rt_link.le_next;
      if ((rt->rt_flags == RTF_UP) && (rt->rt_expire < now))
{
  // if a valid route has expired, purge all packets from
  // send buffer and invalidate the route.
  assert (rt->rt_hops != INFINITY2);
  while ((p = rqueue.deque (rt->rt_dst)))
    {
#ifdef DEBUG
      fprintf (stderr, "%s: calling drop()\n", __FUNCTION__);
#endif // DEBUG
      drop (p, DROP_RTR_NO_ROUTE);
    }
  rt->rt_seqno++;
  assert (rt->rt_seqno % 2);
  rt_down (rt);
}
      else if (rt->rt_flags == RTF_UP)
{
  // If the route is not expired,
  // and there are packets in the sendbuffer waiting,
  // forward them. This should not be needed, but this extra
  // check does no harm.
  assert (rt->rt_hops != INFINITY2);
  while ((p = rqueue.deque (rt->rt_dst)))
    {
      forward (rt, p, delay);
      delay += ARP_DELAY;
    }
}
      else if (rqueue.find (rt->rt_dst))
// If the route is down and
// if there is a packet for this destination waiting in
// the sendbuffer, then send out route request. sendRequest
// will check whether it is time to really send out request
// or not.
// This may not be crucial to do it here, as each generated
// packet will do a sendRequest anyway.

sendRequest (rt->rt_dst);
    }

}

/*
  Packet Reception Routines
*/

void
AODV::recv (Packet * p, Handler *)
{
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);

  assert (initialized ());
  //assert(p->incoming == 0);
  // XXXXX NOTE: use of incoming flag has been depracated; In order to track
direction of pkt flow, direction_ in hdr_cmn is used instead. see packet.h for
details.

  if (ch->ptype () == PT_AODV)
    {
      ih->ttl_ -= 1;
      recvAODV (p);
```

```
      return;
    }

  /*
   *  Must be a packet I'm originating...
   */
  if ((ih->saddr () == index) && (ch->num_forwards () == 0))
    {
      /*
       * Add the IP Header
       */
      ch->size () += IP_HDR_LEN;
      // Added by Parag Dadhania && John Novatnack to handle broadcasting
      if ((u_int32_t) ih->daddr () != IP_BROADCAST)
ih->ttl_ = NETWORK_DIAMETER;
    }
  /*
   *  I received a packet that I sent.  Probably
   *  a routing loop.
   */
  else if (ih->saddr () == index)
    {
      drop (p, DROP_RTR_ROUTE_LOOP);
      return;
    }
  /*
   *  Packet I'm forwarding...
   */
  else
    {
      /*
       *  Check the TTL.  If it is zero, then discard.
       */
      if (--ih->ttl_ == 0)
{
  drop (p, DROP_RTR_TTL);
  return;
}
    }
// Added by Parag Dadhania && John Novatnack to handle broadcasting
  if ((u_int32_t) ih->daddr () != IP_BROADCAST)
    rt_resolve (p);
  else
    forward ((aodv_rt_entry *) 0, p, NO_DELAY);
}


void
AODV::recvAODV (Packet * p)
{
  struct hdr_aodv *ah = HDR_AODV (p);
  struct hdr_ip *ih = HDR_IP (p);

  assert (ih->sport () == RT_PORT);
  assert (ih->dport () == RT_PORT);

  /*
   * Incoming Packets.
   */
  switch (ah->ah_type)
    {

    case AODVTYPE_RREQ:
      recvRequest (p);
      break;

    case AODVTYPE_RREP:
      recvReply (p);
      break;

    case AODVTYPE_RERR:
      recvError (p);
      break;

    case AODVTYPE_HELLO:
      recvHello (p);
      break;
```

```
    default:
      fprintf (stderr, "Invalid AODV type (%x)\n", ah->ah_type);
      exit (1);
    }

}


void
AODV::recvRequest (Packet * p)
{
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_request *rq = HDR_AODV_REQUEST (p);
  aodv_rt_entry *rt;

  /*
   * Drop if:
   *      - I'm the source
   *      - I recently heard this request.
   */

  if (rq->rq_src == index)
    {
#ifdef DEBUG
      fprintf (stderr, "%s: got my own REQUEST\n", __FUNCTION__);
#endif // DEBUG
      Packet::free (p);
      return;
    }

  if (id_lookup (rq->rq_src, rq->rq_bcast_id))
    {

#ifdef DEBUG
      fprintf (stderr, "%s: discarding request\n", __FUNCTION__);
#endif // DEBUG

      Packet::free (p);
      return;
    }

  /*
   * Cache the broadcast ID
   */
  id_insert (rq->rq_src, rq->rq_bcast_id);



  /*
   * We are either going to forward the REQUEST or generate a
   * REPLY. Before we do anything, we make sure that the REVERSE
   * route is in the route table.
   */
  aodv_rt_entry *rt0; // rt0 is the reverse route

  rt0 = rtable.rt_lookup (rq->rq_src);
  if (rt0 == 0)
    { /* if not in the route table */
      // create an entry for the reverse route.
      rt0 = rtable.rt_add (rq->rq_src);
    }

  rt0->rt_expire = max (rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE));

  if ((rq->rq_src_seqno > rt0->rt_seqno) ||
      ((rq->rq_src_seqno == rt0->rt_seqno) &&
       (rq->rq_hop_count < rt0->rt_hops)))
    {
      // If we have a fresher seq no. or lesser #hops for the
      // same seq no., update the rt entry. Else don't bother.
      rt_update (rt0, rq->rq_src_seqno, rq->rq_hop_count, ih->saddr (),
 max (rt0->rt_expire, (CURRENT_TIME + REV_ROUTE_LIFE)));
      if (rt0->rt_req_timeout > 0.0)
{
  // Reset the soft state and
  // Set expiry time to CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT
```

```
  // This is because route is used in the forward direction,
  // but only sources get benefited by this change
  rt0->rt_req_cnt = 0;
  rt0->rt_req_timeout = 0.0;
  rt0->rt_req_last_ttl = rq->rq_hop_count;
  rt0->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
}

      /* Find out whether any buffered packet can benefit from the
       * reverse route.
       * May need some change in the following code - Mahesh 09/11/99
       */
      assert (rt0->rt_flags == RTF_UP);
      Packet *buffered_pkt;
      while ((buffered_pkt = rqueue.deque (rt0->rt_dst)))
{
  if (rt0 && (rt0->rt_flags == RTF_UP))
    {
      assert (rt0->rt_hops != INFINITY2);
      forward (rt0, buffered_pkt, NO_DELAY);
    }
}
    }
  // End for putting reverse route in rt table


  /*
   * We have taken care of the reverse route stuff.
   * Now see whether we can send a route reply.
   */

  //sandy
  //first we see if there are service request extensions present
  if (rq->rq_sreq.sreqp.sq_type == AODVTYPE_SREQ_PORT
      || rq->rq_sreq.srequ.sq_type == AODVTYPE_SREQ_URL)
    {

      aodv_svc_entry *st;
      if (rq->rq_sreq.sreqp.sq_type == AODVTYPE_SREQ_PORT)
st = stable.st_lookup (rq->rq_sreq.sreqp.sqp_port);
      else
st = stable.st_lookup (rq->rq_sreq.srequ.squ_serv_string);
      if (st == NULL && rq->rq_dst == -1) //-1 instead of 0
{
  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  rq->rq_hop_count += 1;
  forward ((aodv_rt_entry *) 0, p, DELAY);
  return;
}
      else
{
  hdr_aodv_srep srep;
  srep.sp_type = AODVTYPE_SREP;
  srep.sp_length =
    sizeof (u_int16_t) + 2 * sizeof (u_int8_t) + (MAX_LENGTH_URL +
  24) * sizeof (char);
  if (st)
    srep.sp_lifetime = (u_int16_t) (st->expire - CURRENT_TIME);
  else
    srep.sp_lifetime = SBINDLTIME; //lifetime set by processing node???
  if (rq->rq_dst != -1)
    {
      rt = rtable.rt_lookup (rq->rq_dst);
      // First check if I am the destination ..
      if (rq->rq_dst == index)
{

#ifdef DEBUG
  fprintf (stderr, "%d - %s: destination sending reply\n",
   index, __FUNCTION__);
#endif // DEBUG


  // Just to be safe, I use the max. Somebody may have
  // incremented the dst seqno.
  seqno = max (seqno, rq->rq_dst_seqno) + 1;
```

```
  if (seqno % 2)
    seqno++;

  //preparing service url for reply
  srep.sp_lifetime = SBINDLTIME;
  strcpy (srep.sp_url, "serivce:");
  strcat (srep.sp_url, st->serv_string);
  strcat (srep.sp_url, "://");
  strcat (srep.sp_url, itoa (index));
  strcat (srep.sp_url, ":");
  strcat (srep.sp_url, itoa (st->port));

  srep.sp_url_length = strlen (srep.sp_url);

  sendSvcReply (rq->rq_src,// IP Destination
1,// Hop Count
index,// Dest IP Address
seqno,// Dest Sequence Num
MY_ROUTE_TIMEOUT,// Lifetime
rq->rq_timestamp,// timestamp
srep); //service req

  Packet::free (p);
  return;
}

      // I am not the destination, but I may have a fresh enough route.

      else if (rt && (rt->rt_hops != INFINITY2) &&
       (rt->rt_seqno >= rq->rq_dst_seqno))
{

  //assert (rt->rt_flags == RTF_UP);
  assert (rq->rq_dst == rt->rt_dst);

  strcpy (srep.sp_url, "serivce:");
  if (st)
    strcat (srep.sp_url, st->serv_string);
  else
    strcat (srep.sp_url, rq->rq_sreq.srequ.squ_serv_string);
  strcat (srep.sp_url, "://");
  strcat (srep.sp_url, itoa (rq->rq_dst));
  strcat (srep.sp_url, ":");
  if (st)
    strcat (srep.sp_url, itoa (st->port));
  else
    strcat (srep.sp_url,
    rq->rq_sreq.srequ.squ_serv_predicate);

  srep.sp_url_length = strlen (srep.sp_url);
  //assert ((rt->rt_seqno%2) == 0);    // is the seqno even?
  sendSvcReply (rq->rq_src,
rt->rt_hops + 1,
rq->rq_dst,
rt->rt_seqno,
(u_int32_t) (rt->rt_expire - CURRENT_TIME),
//             rt->rt_expire - CURRENT_TIME,
rq->rq_timestamp, srep);
  // Insert nexthops to RREQ source and RREQ destination in the
  // precursor lists of destination and source respectively
  rt->pc_insert (rt0->rt_nexthop); // nexthop to RREQ source
  rt0->pc_insert (rt->rt_nexthop); // nexthop to RREQ destination

  Packet::free (p);
  return;
}
    }
  if (st != NULL)
    {
      //if valid route to st->dest,prepare srep & send
      if (st->dest == index)
{
  seqno = max (seqno, rq->rq_dst_seqno) + 1;
  if (seqno % 2)
    seqno++;

  srep.sp_lifetime = SBINDLTIME;
```

```
  strcpy (srep.sp_url, "serivce:");
  strcat (srep.sp_url, st->serv_string);
  strcat (srep.sp_url, "://");
  strcat (srep.sp_url, itoa (index));
  strcat (srep.sp_url, ":");
  strcat (srep.sp_url, itoa (st->port));

  srep.sp_url_length = strlen (srep.sp_url);

  sendSvcReply (rq->rq_src,// IP Destination
1,// Hop Count
index,// Dest IP Address
seqno,// Dest Sequence Num
MY_ROUTE_TIMEOUT,// Lifetime
rq->rq_timestamp,// timestamp
srep); //service req

  Packet::free (p);
  return;
}

      rt = rtable.rt_lookup (st->dest);
      if (rt && (rt->rt_hops != INFINITY2)
  && st->expire > CURRENT_TIME)
{

  //assert (rt->rt_flags == RTF_UP);
  assert (st->dest == rt->rt_dst);

  strcpy (srep.sp_url, "serivce:");
  strcat (srep.sp_url, st->serv_string);
  strcat (srep.sp_url, "://");
  strcat (srep.sp_url, itoa (st->dest));
  strcat (srep.sp_url, ":");
  strcat (srep.sp_url, itoa (st->port));

  srep.sp_url_length = strlen (srep.sp_url);
  //assert ((rt->rt_seqno%2) == 0);    // is the seqno even?
  sendSvcReply (rq->rq_src,
rt->rt_hops + 1,
st->dest,
rt->rt_seqno,
(u_int32_t) (rt->rt_expire - CURRENT_TIME),
//             rt->rt_expire - CURRENT_TIME,
rq->rq_timestamp, srep);
  // Insert nexthops to RREQ source and RREQ destination in the
  // precursor lists of destination and source respectively
  rt->pc_insert (rt0->rt_nexthop); // nexthop to RREQ source
  rt0->pc_insert (rt->rt_nexthop); // nexthop to RREQ destination

  Packet::free (p);
  return;
}
    }

  //set dest. add to resolved add & dest seq. no. to last known seq no &
rebroadcast
  hdr_aodv_sreq_url sreq;

  bzero ((void *) &sreq, sizeof (sreq));
  sreq.sq_type = AODVTYPE_SREQ_URL;
  sreq.sq_length = 2 * sizeof (u_int8_t) + 64 * sizeof (char);
  if (st)
    {
      strcpy (sreq.squ_serv_string, st->serv_string);
      strcpy (sreq.squ_serv_predicate, itoa (st->port));
    }
  else
    {
      strcpy (sreq.squ_serv_string,
      rq->rq_sreq.srequ.squ_serv_string);
      strcpy (sreq.squ_serv_predicate,
      rq->rq_sreq.srequ.squ_serv_predicate);
    }
  sreq.squ_serv_length = strlen (sreq.squ_serv_string);
```

```
  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  if (st && st->expire > CURRENT_TIME)
    rq->rq_dst = st->dest;
  if(rt)rq->rq_dst_seqno = rt->rt_seqno;
  rq->rq_hop_count += 1;
  rq->rq_sreq.srequ = sreq;
  forward ((aodv_rt_entry *) 0, p, DELAY);
  return;
}
    }

  rt = rtable.rt_lookup (rq->rq_dst);

  // First check if I am the destination ..

  if (rq->rq_dst == index)
    {

#ifdef DEBUG
      fprintf (stderr, "%d - %s: destination sending reply\n",
       index, __FUNCTION__);
#endif // DEBUG


      // Just to be safe, I use the max. Somebody may have
      // incremented the dst seqno.
      seqno = max (seqno, rq->rq_dst_seqno) + 1;
      if (seqno % 2)
seqno++;

      sendReply (rq->rq_src,// IP Destination
 1,// Hop Count
 index,// Dest IP Address
 seqno,// Dest Sequence Num
 MY_ROUTE_TIMEOUT,// Lifetime
 rq->rq_timestamp); // timestamp

      Packet::free (p);
    }

  // I am not the destination, but I may have a fresh enough route.

  else if (rt && (rt->rt_hops != INFINITY2) &&
   (rt->rt_seqno >= rq->rq_dst_seqno))
    {

      //assert (rt->rt_flags == RTF_UP);
      assert (rq->rq_dst == rt->rt_dst);
      //assert ((rt->rt_seqno%2) == 0);    // is the seqno even?
      sendReply (rq->rq_src,
 rt->rt_hops + 1,
 rq->rq_dst,
 rt->rt_seqno, (u_int32_t) (rt->rt_expire - CURRENT_TIME),
 //             rt->rt_expire - CURRENT_TIME,
 rq->rq_timestamp);
      // Insert nexthops to RREQ source and RREQ destination in the
      // precursor lists of destination and source respectively
      rt->pc_insert (rt0->rt_nexthop); // nexthop to RREQ source
      rt0->pc_insert (rt->rt_nexthop); // nexthop to RREQ destination

#ifdef RREQ_GRAT_RREP

      sendReply (rq->rq_dst,
 rq->rq_hop_count,
 rq->rq_src,
 rq->rq_src_seqno, (u_int32_t) (rt->rt_expire - CURRENT_TIME),
 //             rt->rt_expire - CURRENT_TIME,
 rq->rq_timestamp);
#endif

// TODO: send grat RREP to dst if G flag set in RREQ using rq->rq_src_seqno,
rq->rq_hop_counT

// DONE: Included gratuitous replies to be sent as per IETF aodv draft
specification. As of now, G flag has not been dynamically used and is always set
or reset in aodv-packet.h --- Anant Utgikar, 09/16/02.
```

```
      Packet::free (p);
    }
  /*
   * Can't reply. So forward the  Route Request
   */
  else
    {
      ih->saddr () = index;
      ih->daddr () = IP_BROADCAST;
      rq->rq_hop_count += 1;
      // Maximum sequence number seen en route
      if (rt)
rq->rq_dst_seqno = max (rt->rt_seqno, rq->rq_dst_seqno);
      forward ((aodv_rt_entry *) 0, p, DELAY);
    }

}


void
AODV::recvReply (Packet * p)
{
//struct hdr_cmn *ch = HDR_CMN(p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_reply *rp = HDR_AODV_REPLY (p);
  aodv_rt_entry *rt;
  char suppress_reply = 0;
  double delay = 0.0;

#ifdef DEBUG
  fprintf (stderr, "%d - %s: received a REPLY\n", index, __FUNCTION__);
#endif // DEBUG


  /*
   *  Got a reply. So reset the "soft state" maintained for
   *  route requests in the request table. We don't really have
   *  have a separate request table. It is just a part of the
   *  routing table itself.
   */
  // Note that rp_dst is the dest of the data packets, not the
  // the dest of the reply, which is the src of the data packets.

  rt = rtable.rt_lookup (rp->rp_dst);

  /*
   *  If I don't have a rt entry to this host... adding
   */
  if (rt == 0)
    {
      rt = rtable.rt_add (rp->rp_dst);
    }

  /*
   * Add a forward route table entry... here I am following
   * Perkins-Royer AODV paper almost literally - SRD 5/99
   */

  if ((rt->rt_seqno < rp->rp_dst_seqno) || // newer route
      ((rt->rt_seqno == rp->rp_dst_seqno) &&
       (rt->rt_hops > rp->rp_hop_count)))
    { // shorter or better route

      // Update the rt entry
      rt_update (rt, rp->rp_dst_seqno, rp->rp_hop_count,
 rp->rp_src, CURRENT_TIME + rp->rp_lifetime);

      // reset the soft state
      rt->rt_req_cnt = 0;
      rt->rt_req_timeout = 0.0;
      rt->rt_req_last_ttl = rp->rp_hop_count;

      if (ih->daddr () == index)
{ // If I am the original source
  // Update the route discovery latency statistics
  // rp->rp_timestamp is the time of request origination
```

```
  rt->rt_disc_latency[rt->hist_indx] =
    (CURRENT_TIME - rp->rp_timestamp) / (double) rp->rp_hop_count;
  // increment indx for next time
  rt->hist_indx = (rt->hist_indx + 1) % MAX_HISTORY;
}

      /*
       * Send all packets queued in the sendbuffer destined for
       * this destination.
       * XXX - observe the "second" use of p.
       */
      Packet *buf_pkt;
      while ((buf_pkt = rqueue.deque (rt->rt_dst)))
{
  if (rt->rt_hops != INFINITY2)
    {
      assert (rt->rt_flags == RTF_UP);
      // Delay them a little to help ARP. Otherwise ARP
      // may drop packets. -SRD 5/23/99
      forward (rt, buf_pkt, delay);
      delay += ARP_DELAY;
    }
}
    }
  else
    {
      suppress_reply = 1;
    }

//sandy
  if (rp->srep.sp_type == AODVTYPE_SREP)
    {
      //st=stable.lookup, add if not present or update if present and lifetime
is more than whatis present
      aodv_svc_entry *st;
      char serv_string[32];
      u_int16_t port;
      extract_serv_param (rp->srep.sp_url, serv_string, &port);
      st = stable.st_lookup (serv_string);
      if (!st)
stable.st_add (port, serv_string, rp->rp_dst,
rp->srep.sp_lifetime,rp->rp_hop_count);
      else
{
  if (((st->expire - CURRENT_TIME) < rp->srep.sp_lifetime) &&
(st->hops>rp->rp_hop_count))
    {
      stable.st_delete (st);
      stable.st_add (port, serv_string, rp->rp_dst,
     rp->srep.sp_lifetime,rp->rp_hop_count);
    }
  else //better binding available
    {
      if (st->dest == index)
rp->srep.sp_lifetime = SBINDLTIME;
      else
rp->srep.sp_lifetime =
  (u_int16_t) (st->expire - CURRENT_TIME);
      strcpy (rp->srep.sp_url, "serivce:");
      strcat (rp->srep.sp_url, st->serv_string);
      strcat (rp->srep.sp_url, "://");
      strcat (rp->srep.sp_url, itoa (st->dest));
      strcat (rp->srep.sp_url, ":");
      strcat (rp->srep.sp_url, itoa (st->port));

      rp->srep.sp_url_length = strlen (rp->srep.sp_url);
      rp->rp_dst = st->dest;
    }
}
      //if any agent started service discovery, inform a service is available
      svc_announce (port, serv_string, rp->rp_dst);
    }

  /*
   * If reply is for me, discard it.
   */
```

```
  if (ih->daddr () == index || suppress_reply)
    {
      Packet::free (p);
    }
  /*
   * Otherwise, forward the Route Reply.
   */
  else
    {
      // Find the rt entry
      aodv_rt_entry *rt0 = rtable.rt_lookup (ih->daddr ());
      // If the rt is up, forward
      if (rt0 && (rt0->rt_hops != INFINITY2))
{
  assert (rt0->rt_flags == RTF_UP);
  rp->rp_hop_count += 1;
  rp->rp_src = index;
  forward (rt0, p, NO_DELAY);
  // Insert the nexthop towards the RREQ source to
  // the precursor list of the RREQ destination
  rt->pc_insert (rt0->rt_nexthop); // nexthop to RREQ source

}
      else
{
  // I don't know how to forward .. drop the reply.
#ifdef DEBUG
  fprintf (stderr, "%s: dropping Route Reply\n", __FUNCTION__);
#endif // DEBUG
  drop (p, DROP_RTR_NO_ROUTE);
}
    }
}


void
AODV::recvError (Packet * p)
{
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_error *re = HDR_AODV_ERROR (p);
  aodv_rt_entry *rt;
  u_int8_t i;
  Packet *rerr = Packet::alloc ();
  struct hdr_aodv_error *nre = HDR_AODV_ERROR (rerr);

  nre->DestCount = 0;

  for (i = 0; i < re->DestCount; i++)
    {
      // For each unreachable destination
      rt = rtable.rt_lookup (re->unreachable_dst[i]);
      if (rt && (rt->rt_hops != INFINITY2) &&
  (rt->rt_nexthop == ih->saddr ()) &&
  (rt->rt_seqno <= re->unreachable_dst_seqno[i]))
{
  assert (rt->rt_flags == RTF_UP);
  assert ((rt->rt_seqno % 2) == 0); // is the seqno even?
#ifdef DEBUG
  fprintf (stderr, "%s(%f): %d\t(%d\t%u\t%d)\t(%d\t%u\t%d)\n",
   __FUNCTION__, CURRENT_TIME, index, rt->rt_dst,
   rt->rt_seqno, rt->rt_nexthop, re->unreachable_dst[i],
   re->unreachable_dst_seqno[i], ih->saddr ());
#endif // DEBUG
  rt->rt_seqno = re->unreachable_dst_seqno[i];
  rt_down (rt);

  // Not sure whether this is the right thing to do
  Packet *pkt;
  while ((pkt = ifqueue->filter (ih->saddr ())))
    {
      drop (pkt, DROP_RTR_MAC_CALLBACK);
    }

  // if precursor list non-empty add to RERR and delete the precursor list
  if (!rt->pc_empty ())
    {
```

```
      nre->unreachable_dst[nre->DestCount] = rt->rt_dst;
      nre->unreachable_dst_seqno[nre->DestCount] = rt->rt_seqno;
      nre->DestCount += 1;
      rt->pc_delete ();
    }
}
    }

  if (nre->DestCount > 0)
    {
#ifdef DEBUG
      fprintf (stderr, "%s(%f): %d\t sending RERR...\n", __FUNCTION__,
       CURRENT_TIME, index);
#endif // DEBUG
      sendError (rerr);
    }
  else
    {
      Packet::free (rerr);
    }

  Packet::free (p);
}


/*
   Packet Transmission Routines
*/

void
AODV::forward (aodv_rt_entry * rt, Packet * p, double delay)
{
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);

  if (ih->ttl_ == 0)
    {

#ifdef DEBUG
      fprintf (stderr, "%s: calling drop()\n", __PRETTY_FUNCTION__);
#endif // DEBUG

      drop (p, DROP_RTR_TTL);
      return;
    }

  if (rt)
    {
      assert (rt->rt_flags == RTF_UP);
      rt->rt_expire = CURRENT_TIME + ACTIVE_ROUTE_TIMEOUT;
      ch->next_hop_ = rt->rt_nexthop;
      ch->addr_type () = NS_AF_INET;
      ch->direction () = hdr_cmn::DOWN; //important: change the packet's
direction
    }
  else
    { // if it is a broadcast packet
      // assert(ch->ptype() == PT_AODV); // maybe a diff pkt type like gaf
      assert (ih->daddr () == (nsaddr_t) IP_BROADCAST);
      ch->addr_type () = NS_AF_NONE;
      ch->direction () = hdr_cmn::DOWN; //important: change the packet's
direction
    }

  if (ih->daddr () == (nsaddr_t) IP_BROADCAST)
    {
      // If it is a broadcast packet
      assert (rt == 0);
      /*
       *  Jitter the sending of broadcast packets by 10ms
       */
      Scheduler::instance ().schedule (target_, p, 0.01 * Random::uniform ());
    }
  else
    { // Not a broadcast packet
      if (delay > 0.0)
{
```

```
  Scheduler::instance ().schedule (target_, p, delay);
}
      else
{
  // Not a broadcast packet, no delay, send immediately
  Scheduler::instance ().schedule (target_, p, 0.);
}
    }

}


void
AODV::sendRequest (nsaddr_t dst)
{
// Allocate a RREQ packet
  Packet *p = Packet::alloc ();
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_request *rq = HDR_AODV_REQUEST (p);
  aodv_rt_entry *rt = rtable.rt_lookup (dst);

  assert (rt);

  /*
   *  Rate limit sending of Route Requests. We are very conservative
   *  about sending out route requests.
   */

  if (rt->rt_flags == RTF_UP)
    {
      assert (rt->rt_hops != INFINITY2);
      Packet::free ((Packet *) p);
      return;
    }

  if (rt->rt_req_timeout > CURRENT_TIME)
    {
      Packet::free ((Packet *) p);
      return;
    }

  // rt_req_cnt is the no. of times we did network-wide broadcast
  // RREQ_RETRIES is the maximum number we will allow before
  // going to a long timeout.

  if (rt->rt_req_cnt > RREQ_RETRIES)
    {
      rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
      rt->rt_req_cnt = 0;
      Packet *buf_pkt;
      while ((buf_pkt = rqueue.deque (rt->rt_dst)))
{
  drop (buf_pkt, DROP_RTR_NO_ROUTE);
}
      Packet::free ((Packet *) p);
      return;
    }

#ifdef DEBUG
  fprintf (stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
   ++route_request, index, rt->rt_dst);
#endif // DEBUG

  // Determine the TTL to be used this time.
  // Dynamic TTL evaluation - SRD

  rt->rt_req_last_ttl = max (rt->rt_req_last_ttl, rt->rt_last_hop_count);

  if (0 == rt->rt_req_last_ttl)
    {
      // first time query broadcast
      ih->ttl_ = TTL_START;
    }
  else
    {
      // Expanding ring search.
```

```
      if (rt->rt_req_last_ttl < TTL_THRESHOLD)
ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
      else
{
  // network-wide broadcast
  ih->ttl_ = NETWORK_DIAMETER;
  rt->rt_req_cnt += 1;
}
    }

  // remember the TTL used  for the next time
  rt->rt_req_last_ttl = ih->ttl_;

  // PerHopTime is the roundtrip time per hop for route requests.
  // The factor 2.0 is just to be safe .. SRD 5/22/99
  // Also note that we are making timeouts to be larger if we have
  // done network wide broadcast before.

  rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime (rt);
  if (rt->rt_req_cnt > 0)
    rt->rt_req_timeout *= rt->rt_req_cnt;
  rt->rt_req_timeout += CURRENT_TIME;

  // Don't let the timeout to be too large, however .. SRD 6/8/99
  if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
    rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
  rt->rt_expire = 0;

#ifdef DEBUG
  fprintf (stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
   ++route_request,
   index, rt->rt_dst, rt->rt_req_timeout - CURRENT_TIME);
#endif // DEBUG


  // Fill out the RREQ packet
  // ch->uid() = 0;
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + rq->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_NONE;
  ch->prev_hop_ = index; // AODV hack

  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;

  // Fill up some more fields.
  rq->rq_type = AODVTYPE_RREQ;
  rq->rq_hop_count = 1;
  rq->rq_bcast_id = bid++;
  rq->rq_dst = dst;
  rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
  rq->rq_src = index;
  seqno += 2;
  assert ((seqno % 2) == 0);
  rq->rq_src_seqno = seqno;
  rq->rq_timestamp = CURRENT_TIME;

  Scheduler::instance ().schedule (target_, p, 0.);

}

//sandy
void
AODV::sendSvcRequest (nsaddr_t dst, hdr_aodv_sreq sreq)
{
// Allocate a RREQ packet
  Packet *p = Packet::alloc ();
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_request *rq = HDR_AODV_REQUEST (p);
  aodv_rt_entry *rt = rtable.rt_lookup (dst);

  rq->rq_sreq = sreq; //sandy
```

```
  assert (rt);

  /*
   *  Rate limit sending of Route Requests. We are very conservative
   *  about sending out route requests.
   */

  if (rt->rt_flags == RTF_UP)
    {
      assert (rt->rt_hops != INFINITY2);
      Packet::free ((Packet *) p);
      return;
    }

  if (rt->rt_req_timeout > CURRENT_TIME)
    {
      Packet::free ((Packet *) p);
      return;
    }

  // rt_req_cnt is the no. of times we did network-wide broadcast
  // RREQ_RETRIES is the maximum number we will allow before
  // going to a long timeout.

  if (rt->rt_req_cnt > RREQ_RETRIES)
    {
      rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
      rt->rt_req_cnt = 0;
      Packet *buf_pkt;
      while ((buf_pkt = rqueue.deque (rt->rt_dst)))
{
  drop (buf_pkt, DROP_RTR_NO_ROUTE);
}
      Packet::free ((Packet *) p);
      return;
    }

#ifdef DEBUG
  fprintf (stderr, "(%2d) - %2d sending Route Request, dst: %d\n",
   ++route_request, index, rt->rt_dst);
#endif // DEBUG

  // Determine the TTL to be used this time.
  // Dynamic TTL evaluation - SRD

  rt->rt_req_last_ttl = max (rt->rt_req_last_ttl, rt->rt_last_hop_count);

  if (0 == rt->rt_req_last_ttl)
    {
      // first time query broadcast
      ih->ttl_ = TTL_START;
    }
  else
    {
      // Expanding ring search.
      if (rt->rt_req_last_ttl < TTL_THRESHOLD)
ih->ttl_ = rt->rt_req_last_ttl + TTL_INCREMENT;
      else
{
  // network-wide broadcast
  ih->ttl_ = NETWORK_DIAMETER;
  rt->rt_req_cnt += 1;
}
    }

  // remember the TTL used  for the next time
  rt->rt_req_last_ttl = ih->ttl_;

  // PerHopTime is the roundtrip time per hop for route requests.
  // The factor 2.0 is just to be safe .. SRD 5/22/99
  // Also note that we are making timeouts to be larger if we have
  // done network wide broadcast before.

  rt->rt_req_timeout = 2.0 * (double) ih->ttl_ * PerHopTime (rt);
  if (rt->rt_req_cnt > 0)
    rt->rt_req_timeout *= rt->rt_req_cnt;
  rt->rt_req_timeout += CURRENT_TIME;
```

```
  // Don't let the timeout to be too large, however .. SRD 6/8/99
  if (rt->rt_req_timeout > CURRENT_TIME + MAX_RREQ_TIMEOUT)
    rt->rt_req_timeout = CURRENT_TIME + MAX_RREQ_TIMEOUT;
  rt->rt_expire = 0;

#ifdef DEBUG
  fprintf (stderr, "(%2d) - %2d sending Route Request, dst: %d, tout %f ms\n",
   ++route_request,
   index, rt->rt_dst, rt->rt_req_timeout - CURRENT_TIME);
#endif // DEBUG


  // Fill out the RREQ packet
  // ch->uid() = 0;
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + rq->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_NONE;
  ch->prev_hop_ = index; // AODV hack

  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;

  // Fill up some more fields.
  rq->rq_type = AODVTYPE_RREQ;
  rq->rq_hop_count = 1;
  rq->rq_bcast_id = bid++;
  rq->rq_dst = dst;
  rq->rq_dst_seqno = (rt ? rt->rt_seqno : 0);
  rq->rq_src = index;
  seqno += 2;
  assert ((seqno % 2) == 0);
  rq->rq_src_seqno = seqno;
  rq->rq_timestamp = CURRENT_TIME;

  Scheduler::instance ().schedule (target_, p, 0.);

}

void
AODV::sendReply (nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
 u_int32_t rpseq, u_int32_t lifetime, double timestamp)
{
  Packet *p = Packet::alloc ();
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_reply *rp = HDR_AODV_REPLY (p);
  aodv_rt_entry *rt = rtable.rt_lookup (ipdst);

#ifdef DEBUG
  fprintf (stderr, "sending Reply from %d at %.2f\n", index,
   Scheduler::instance ().clock ());
#endif // DEBUG
  assert (rt);

  rp->rp_type = AODVTYPE_RREP;
  //rp->rp_flags = 0x00;
  rp->rp_hop_count = hop_count;
  rp->rp_dst = rpdst;
  rp->rp_dst_seqno = rpseq;
  rp->rp_src = index;
  rp->rp_lifetime = lifetime;
  rp->rp_timestamp = timestamp;

  // ch->uid() = 0;
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + rp->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_INET;
  ch->next_hop_ = rt->rt_nexthop;
  ch->prev_hop_ = index; // AODV hack
  ch->direction () = hdr_cmn::DOWN;
```

```
  ih->saddr () = index;
  ih->daddr () = ipdst;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;
  ih->ttl_ = NETWORK_DIAMETER;

  Scheduler::instance ().schedule (target_, p, 0.);

}

//sandy
void
AODV::sendSvcReply (nsaddr_t ipdst, u_int32_t hop_count, nsaddr_t rpdst,
    u_int32_t rpseq, u_int32_t lifetime, double timestamp,
    hdr_aodv_srep srep)
{
  Packet *p = Packet::alloc ();
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_reply *rp = HDR_AODV_REPLY (p);
  aodv_rt_entry *rt = rtable.rt_lookup (ipdst);

#ifdef DEBUG
  fprintf (stderr, "sending Reply from %d at %.2f\n", index,
   Scheduler::instance ().clock ());
#endif // DEBUG
  assert (rt);

  rp->rp_type = AODVTYPE_RREP;
  //rp->rp_flags = 0x00;
  rp->rp_hop_count = hop_count;
  rp->rp_dst = rpdst;
  rp->rp_dst_seqno = rpseq;
  rp->rp_src = index;
  rp->rp_lifetime = lifetime;
  rp->rp_timestamp = timestamp;
  rp->srep = srep;

  // ch->uid() = 0;
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + rp->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_INET;
  ch->next_hop_ = rt->rt_nexthop;
  ch->prev_hop_ = index; // AODV hack
  ch->direction () = hdr_cmn::DOWN;

  ih->saddr () = index;
  ih->daddr () = ipdst;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;
  ih->ttl_ = NETWORK_DIAMETER;

  Scheduler::instance ().schedule (target_, p, 0.);

}


void
AODV::sendError (Packet * p, bool jitter)
{
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_error *re = HDR_AODV_ERROR (p);

#ifdef ERROR
  fprintf (stderr, "sending Error from %d at %.2f\n", index,
   Scheduler::instance ().clock ());
#endif // DEBUG

  re->re_type = AODVTYPE_RERR;
  //re->reserved[0] = 0x00; re->reserved[1] = 0x00;
  // DestCount and list of unreachable destinations are already filled

  // ch->uid() = 0;
```

```
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + re->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_NONE;
  ch->next_hop_ = 0;
  ch->prev_hop_ = index; // AODV hack
  ch->direction () = hdr_cmn::DOWN; //important: change the packet's direction

  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;
  ih->ttl_ = 1;

  // Do we need any jitter? Yes
  if (jitter)
    Scheduler::instance ().schedule (target_, p, 0.01 * Random::uniform ());
  else
    Scheduler::instance ().schedule (target_, p, 0.0);

}


/*
   Neighbor Management Functions
*/

void
AODV::sendHello ()
{
  Packet *p = Packet::alloc ();
  struct hdr_cmn *ch = HDR_CMN (p);
  struct hdr_ip *ih = HDR_IP (p);
  struct hdr_aodv_reply *rh = HDR_AODV_REPLY (p);

#ifdef DEBUG
  fprintf (stderr, "sending Hello from %d at %.2f\n", index,
   Scheduler::instance ().clock ());
#endif // DEBUG

  rh->rp_type = AODVTYPE_HELLO;
  //rh->rp_flags = 0x00;
  rh->rp_hop_count = 1;
  rh->rp_dst = index;
  rh->rp_dst_seqno = seqno;
  rh->rp_lifetime = (1 + ALLOWED_HELLO_LOSS) * HELLO_INTERVAL;

  // ch->uid() = 0;
  ch->ptype () = PT_AODV;
  ch->size () = IP_HDR_LEN + rh->size ();
  ch->iface () = -2;
  ch->error () = 0;
  ch->addr_type () = NS_AF_NONE;
  ch->prev_hop_ = index; // AODV hack

  ih->saddr () = index;
  ih->daddr () = IP_BROADCAST;
  ih->sport () = RT_PORT;
  ih->dport () = RT_PORT;
  ih->ttl_ = 1;

  Scheduler::instance ().schedule (target_, p, 0.0);
}


void
AODV::recvHello (Packet * p)
{
//struct hdr_ip *ih = HDR_IP(p);
  struct hdr_aodv_reply *rp = HDR_AODV_REPLY (p);
  AODV_Neighbor *nb;

  nb = nb_lookup (rp->rp_dst);
  if (nb == 0)
    {
      nb_insert (rp->rp_dst);
```

```
    }
  else
    {
      nb->nb_expire = CURRENT_TIME +
(1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
    }

  Packet::free (p);
}

void
AODV::nb_insert (nsaddr_t id)
{
  AODV_Neighbor *nb = new AODV_Neighbor (id);

  assert (nb);
  nb->nb_expire = CURRENT_TIME + (1.5 * ALLOWED_HELLO_LOSS * HELLO_INTERVAL);
  LIST_INSERT_HEAD (&nbhead, nb, nb_link);
  seqno += 2; // set of neighbors changed
  assert ((seqno % 2) == 0);
}


AODV_Neighbor *
AODV::nb_lookup (nsaddr_t id)
{
  AODV_Neighbor *nb = nbhead.lh_first;

  for (; nb; nb = nb->nb_link.le_next)
    {
      if (nb->nb_addr == id)
break;
    }
  return nb;
}


/*
 * Called when we receive *explicit* notification that a Neighbor
 * is no longer reachable.
 */
void
AODV::nb_delete (nsaddr_t id)
{
  AODV_Neighbor *nb = nbhead.lh_first;

  log_link_del (id);
  seqno += 2; // Set of neighbors changed
  assert ((seqno % 2) == 0);

  for (; nb; nb = nb->nb_link.le_next)
    {
      if (nb->nb_addr == id)
{
  LIST_REMOVE (nb, nb_link);
  delete nb;
  break;
}
    }

  handle_link_failure (id);

}


/*
 * Purges all timed-out Neighbor Entries - runs every
 * HELLO_INTERVAL * 1.5 seconds.
 */
void
AODV::nb_purge ()
{
  AODV_Neighbor *nb = nbhead.lh_first;
  AODV_Neighbor *nbn;
  double now = CURRENT_TIME;

  for (; nb; nb = nbn)
```

```
    {
      nbn = nb->nb_link.le_next;
      if (nb->nb_expire <= now)
{
  nb_delete (nb->nb_addr);
}
    }

}

//sandy
void
AODV::svc_discover (u_int16_t port, char serv_string[32])
{
  aodv_svc_entry *st = NULL;
  hdr_aodv_sreq sreq;
  char c[100];
  Tcl & tcl = Tcl::instance ();

  if (strcmp (serv_string, ""))
    {
      sreq.srequ.sq_type = AODVTYPE_SREQ_URL;
      sreq.srequ.sq_length =
2 * sizeof (u_int16_t) + (MAX_LENGTH_URL + 24) * sizeof (char);
      sreq.srequ.squ_serv_length = strlen (serv_string);
      strcpy (sreq.srequ.squ_serv_string, serv_string);
      strcpy (sreq.srequ.squ_serv_predicate, itoa (port));
    }
  else if (port > 0)
    {
      sreq.sreqp.sq_type = AODVTYPE_SREQ_PORT;
      sreq.sreqp.sq_length = sizeof (u_int16_t);
      sreq.sreqp.sqp_port = port;
    }
  else
    {
      tcl.evalf ("puts \"ERROR: bad port no. or service request\"");
      return;
    }


  //check if service binding exists
  if (port > 0)
    st = stable.st_lookup (port);
  if (!st)
    st = stable.st_lookup (serv_string);

  if (st)
    {
      if (st->expire > CURRENT_TIME) //if valid....announce ``service available"
{
  sprintf (c,
   "puts \"%f\t%d: Service %s available on port no. %d at machine %d\"",
   CURRENT_TIME, index, st->serv_string, st->port, st->dest);
  tcl.eval (c);
  return;
}
      else
sendSvcRequest (st->dest, sreq);
    }
  else
    sendSvcRequest (-1, sreq); //set dst as -1 and send

  //add entry to service discovery table
  servdiscoverQ *temp = new servdiscoverQ;
  temp->port = port;
  strcpy (temp->serv_string, serv_string);
  temp->expire = CURRENT_TIME + SDISEXTIME;
  temp->next = NULL;
  if (sdqtail)
    sdqtail->next = temp;
  else
    sdqhead = sdqtail = temp;

}

void
```

```
AODV::svc_announce (u_int16_t port, char serv_string[32], nsaddr_t id)
{
  servdiscoverQ *temp, *temp1 = NULL;
  char c[100];
  Tcl & tcl = Tcl::instance ();
  //check from table if any agent is waiting for this service
  temp = sdqhead;
  while (temp != NULL)
    {
      if (temp->port == port || !strcasecmp (serv_string, temp->serv_string))
{
  sprintf (c,
   "puts \"%f\t%d: Service %s available on port no. %d at machine %d\"",
   CURRENT_TIME, index, serv_string, port, id);
  tcl.evalf (c);
  //instead of priting out statement as shown above a signal can be delivered
to an application so that it can start its data traffic
  if (temp == sdqhead)
    {
      if (sdqhead == sdqtail)
sdqhead = sdqtail = NULL;
      else
sdqhead = sdqhead->next;
    }
  else if (temp == sdqtail)
    {
      sdqtail = temp1;
      sdqtail->next = NULL;
    }
  else
    {
      temp1->next = temp->next;
      temp->next = NULL;
    }
  delete temp;
}
      temp1 = temp;
      temp = temp->next;
    }
}

void
AODV::extract_serv_param (char *url, char *serv_string, u_int16_t * port)
{
  char *temp = rindex (url, ':'), *temp1;
  int n;
  *port = (u_int16_t) atoi (temp + 1);
  temp =::index (url, ':');
  temp1 =::index (temp + 1, ':');
  n = temp1 - temp - 1;
  strncpy (serv_string, temp + 1, n);
  serv_string[n] = '\0';
}

//simple integer to ascii string conversion
char *
itoa (int no)
{
  char *asc, k;
  int i, j;
  asc=(char *)malloc(6*sizeof(char));
  if (no == 0)
    {
      asc[0] = '0';
      asc[1] = '\0';
      return asc;
    }
  for (i = 0; no > 0; i++)
    {
      j = no % 10;
      asc[i] = j + '0';
      no /= 10;
    }
  asc[i] = '\0';
  i--;
  for (j = 0; j < i; j++, i--)
    {
```

```
      k = asc[j];
      asc[j] = asc[i];
      asc[i] = k;
    }
  return asc;
}

void
AODV::svc_purge ()
{
  servdiscoverQ *temp, *temp1 = NULL;
  char c[100];
  Tcl & tcl = Tcl::instance ();

  temp = sdqhead;
  while (temp != NULL)
    {
      if (temp->expire < CURRENT_TIME)
{
  sprintf (c, "puts \"%f\t%d: Service ", CURRENT_TIME, index);
  if (temp->port > 0)
    {
      strcat (c, "at port no. ");
      strcat (c, itoa (temp->port));
    }
  else
    strcat (c, temp->serv_string);
  strcat (c, " couldnot be discovered\"");
  tcl.eval (c);

  if (temp == sdqhead)
    {
      if (sdqhead == sdqtail)
sdqhead = sdqtail = NULL;
      else
sdqhead = sdqhead->next;
    }
  else if (temp == sdqtail)
    {
      sdqtail = temp1;
      sdqtail->next = NULL;
    }
  else
    {
      temp1->next = temp->next;
      temp->next = NULL;
    }
  delete temp;
}
      temp1 = temp;
      temp = temp->next;
    }
  stable.st_purge (index);
}
```

B.3 aodv_packet.h

```
/*Modified and extended by Sandeep Gupta, Indian Institute of Information
  Technology, Allahabad,
  India.
  May, 2003.
*/

/* Copyright (c) 2003 Indian Insitute of Information Technology, Allahabad.
  * All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted.
  *
  * The software is provided "as is" without express or implied warranty of any
  * kind arising out of the use of this software.
*/

/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice
and this permission notice appear in all copies of the software,
derivative works, or modified versions, and any portions thereof,
and that both notices appear in supporting documentation, and that
credit is given to Carnegie Mellon University in all publications
reporting on direct or indirect use of this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights
to redistribute these changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned
by Samir Das and Mahesh Marina, University of Cincinnati. The work was
partially done in Sun Microsystems.
*/


#ifndef __aodv_packet_h__
#define __aodv_packet_h__

//#include <config.h>
//#include "aodv.h"
#define AODV_MAX_ERRORS 100


/* ======================================================================
   Packet Formats...
   ====================================================================== */
#define AODVTYPE_HELLO   0x01
#define AODVTYPE_RREQ    0x02
#define AODVTYPE_RREP    0x04
#define AODVTYPE_RERR    0x08
#define AODVTYPE_RREP_ACK   0x10

//sandy
#define AODVTYPE_SREQ_PORT 0x81
```

```
#define AODVTYPE_SREQ_URL 0x82
#define AODVTYPE_SREP 0x83
#define MAX_LENGTH_URL 255
#define INFLTIME 255

/*
 * AODV Routing Protocol Header Macros
 */
#define HDR_AODV(p) ((struct hdr_aodv*)hdr_aodv::access(p))
#define HDR_AODV_REQUEST(p)   ((struct hdr_aodv_request*)hdr_aodv::access(p))
#define HDR_AODV_REPLY(p) ((struct hdr_aodv_reply*)hdr_aodv::access(p))
#define HDR_AODV_ERROR(p) ((struct hdr_aodv_error*)hdr_aodv::access(p))
#define HDR_AODV_RREP_ACK(p) ((struct hdr_aodv_rrep_ack*)hdr_aodv::access(p))

//sandy

struct hdr_aodv_sreq_port
{
  u_int8_t sq_type; //type
  u_int8_t sq_length; //length of header
  u_int16_t sqp_port; //port number
};

struct hdr_aodv_sreq_url
{
  u_int8_t sq_type;
  u_int8_t sq_length;
  u_int8_t squ_serv_length; //length of service type string
  u_int8_t reserverd;
  char squ_serv_string[32]; //service string
  char squ_serv_predicate[32]; //service predicate
};

struct hdr_aodv_srep
{
  u_int8_t sp_type; //type
  u_int8_t sp_length; //length of header
  u_int16_t sp_lifetime; //lifetime of service/route

  u_int8_t sp_url_length; //length of url
  char sp_url[MAX_LENGTH_URL]; //url
  u_int8_t sp_no_url_auth; //# of autherization blocks
  char sp_auth_block[24]; //authorization block data
};

union hdr_aodv_sreq
{
  hdr_aodv_sreq_port sreqp;
  hdr_aodv_sreq_url srequ;
};

/*
 * General AODV Header - shared by all formats
 */
struct hdr_aodv
{
  u_int8_t ah_type;
  /*
     u_int8_t        ah_reserved[2];
     u_int8_t        ah_hopcount;
   */
  // Header access methods
  static int offset_; // required by PacketHeaderManager
  inline static int &offset ()
  {
    return offset_;
  }
  inline static hdr_aodv *access (const Packet * p)
  {
    return (hdr_aodv *) p->access (offset_);
  }
};

struct hdr_aodv_request
{
  u_int8_t rq_type; // Packet Type
  u_int8_t reserved[2];
```

```
  u_int8_t rq_hop_count; // Hop Count
  u_int32_t rq_bcast_id; // Broadcast ID

  nsaddr_t rq_dst; // Destination IP Address
  u_int32_t rq_dst_seqno; // Destination Sequence Number
  nsaddr_t rq_src; // Source IP Address
  u_int32_t rq_src_seqno; // Source Sequence Number

  double rq_timestamp; // when REQUEST sent;
  // used to compute route discovery latency
  hdr_aodv_sreq rq_sreq; //for service request extensions - sandy

  // This define turns on gratuitous replies- see aodv.cc for implementation contributed by
  // Anant Utgikar, 09/16/02.
  //#define RREQ_GRAT_RREP      0x80

  inline int size ()
  {
    int sz = 0;
    /*
       sz = sizeof(u_int8_t)              // rq_type
       + 2*sizeof(u_int8_t)       // reserved
       + sizeof(u_int8_t)         // rq_hop_count
       + sizeof(double)           // rq_timestamp
       + sizeof(u_int32_t)        // rq_bcast_id
       + sizeof(nsaddr_t)         // rq_dst
       + sizeof(u_int32_t)        // rq_dst_seqno
       + sizeof(nsaddr_t)         // rq_src
       + sizeof(u_int32_t);       // rq_src_seqno
     */
      sz = 7 * sizeof (u_int32_t);
    //sandy
    if (rq_sreq.sreqp.sq_type == AODVTYPE_SREQ_PORT)
        sz += sizeof (hdr_aodv_sreq_port);
    else if (rq_sreq.sreqp.sq_type == AODVTYPE_SREQ_URL)
        sz += sizeof (hdr_aodv_sreq_url);
      assert (sz >= 0);
      return sz;
  }
};

struct hdr_aodv_reply
{
  u_int8_t rp_type; // Packet Type
  u_int8_t reserved[2];
  u_int8_t rp_hop_count; // Hop Count
  nsaddr_t rp_dst; // Destination IP Address
  u_int32_t rp_dst_seqno; // Destination Sequence Number
  nsaddr_t rp_src; // Source IP Address
  double rp_lifetime; // Lifetime

  double rp_timestamp; // when corresponding REQ sent;
  // used to compute route discovery latency
  hdr_aodv_srep srep; //for service reply extensions - sandy

  inline int size ()
  {
    int sz = 0;
    /*
       sz = sizeof(u_int8_t)              // rp_type
       + 2*sizeof(u_int8_t)       // rp_flags + reserved
       + sizeof(u_int8_t)         // rp_hop_count
       + sizeof(double)           // rp_timestamp
       + sizeof(nsaddr_t)         // rp_dst
       + sizeof(u_int32_t)        // rp_dst_seqno
       + sizeof(nsaddr_t)         // rp_src
       + sizeof(u_int32_t);       // rp_lifetime
     */
      sz = 6 * sizeof (u_int32_t);
    if (srep.sp_type == AODVTYPE_SREP)
        sz += sizeof (hdr_aodv_srep);
      assert (sz >= 0);
      return sz;
  }

};
```

```
struct hdr_aodv_error
{
  u_int8_t re_type; // Type
  u_int8_t reserved[2]; // Reserved
  u_int8_t DestCount; // DestCount
  // List of Unreachable destination IP addresses and sequence numbers
  nsaddr_t unreachable_dst[AODV_MAX_ERRORS];
  u_int32_t unreachable_dst_seqno[AODV_MAX_ERRORS];

  inline int size ()
  {
    int sz = 0;
    /*
       sz = sizeof(u_int8_t)              // type
       + 2*sizeof(u_int8_t)       // reserved
       + sizeof(u_int8_t)         // length
       + length*sizeof(nsaddr_t); // unreachable destinations
     */
      sz = (DestCount * 2 + 1) * sizeof (u_int32_t);
      assert (sz);
      return sz;
  }

};

struct hdr_aodv_rrep_ack
{
  u_int8_t rpack_type;
  u_int8_t reserved;
};

// for size calculation of header-space reservation
union hdr_all_aodv
{
  hdr_aodv ah;
  hdr_aodv_request rreq;
  hdr_aodv_reply rrep;
  hdr_aodv_error rerr;
  hdr_aodv_rrep_ack rrep_ack;
};


#endif /* __aodv_packet_h__ */
```

B.4 aodv_rtable.h

```
/*Modified and extended by Sandeep Gupta, Indian Institute of Information
  Technology, Allahabad,
  India.
  May, 2003.
*/

/* Copyright (c) 2003 Indian Insitute of Information Technology, Allahabad.
  * All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted.
  *
  * The software is provided "as is" without express or implied warranty of any
  * kind arising out of the use of this software.
*/

/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice
and this permission notice appear in all copies of the software,
derivative works, or modified versions, and any portions thereof,
and that both notices appear in supporting documentation, and that
credit is given to Carnegie Mellon University in all publications
reporting on direct or indirect use of this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights
to redistribute these changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned
by Samir Das and Mahesh Marina, University of Cincinnati. The work was
partially done in Sun Microsystems.
*/


#ifndef __aodv_rtable_h__
#define __aodv_rtable_h__

#include <assert.h>
#include <sys/types.h>
#include <config.h>
#include <lib/bsd-list.h>
#include <scheduler.h>

#define CURRENT_TIME    Scheduler::instance().clock()
#define INFINITY2        0xff

/*
   AODV Neighbor Cache Entry
*/
class AODV_Neighbor
{
  friend class AODV;
  friend class aodv_rt_entry;
```

```
public:
    AODV_Neighbor (u_int32_t a)
  {
    nb_addr = a;
  }

protected:
    LIST_ENTRY (AODV_Neighbor) nb_link;
  nsaddr_t nb_addr;
  double nb_expire; // ALLOWED_HELLO_LOSS * HELLO_INTERVAL
};

LIST_HEAD (aodv_ncache, AODV_Neighbor);

/*
   AODV Precursor list data structure
*/
class AODV_Precursor
{
  friend class AODV;
  friend class aodv_rt_entry;
public:
    AODV_Precursor (u_int32_t a)
  {
    pc_addr = a;
  }

protected:
    LIST_ENTRY (AODV_Precursor) pc_link;
  nsaddr_t pc_addr; // precursor address
};

LIST_HEAD (aodv_precursors, AODV_Precursor);


/*
  Route Table Entry
*/

class aodv_rt_entry
{
  friend class aodv_rtable;
  friend class AODV;
  friend class LocalRepairTimer;
public:
    aodv_rt_entry ();
   ~aodv_rt_entry ();

  void nb_insert (nsaddr_t id);
  AODV_Neighbor *nb_lookup (nsaddr_t id);

  void pc_insert (nsaddr_t id);
  AODV_Precursor *pc_lookup (nsaddr_t id);
  void pc_delete (nsaddr_t id);
  void pc_delete (void);
  bool pc_empty (void);

  double rt_req_timeout; // when I can send another req
  u_int8_t rt_req_cnt; // number of route requests

protected:
    LIST_ENTRY (aodv_rt_entry) rt_link;

  nsaddr_t rt_dst;
  u_int32_t rt_seqno;
  /* u_int8_t     rt_interface; */
  u_int16_t rt_hops; // hop count
  int rt_last_hop_count; // last valid hop count
  nsaddr_t rt_nexthop; // next hop IP address
  /* list of precursors */
  aodv_precursors rt_pclist;
  double rt_expire; // when entry expires
  u_int8_t rt_flags;

#define RTF_DOWN 0
#define RTF_UP 1
#define RTF_IN_REPAIR 2
```

```
  /*
   *  Must receive 4 errors within 3 seconds in order to mark
   *  the route down.
   u_int8_t        rt_errors;      // error count
   double          rt_error_time;
   #define MAX_RT_ERROR            4       // errors
   #define MAX_RT_ERROR_TIME       3       // seconds
   */

#define MAX_HISTORY 3
  double rt_disc_latency[MAX_HISTORY];
  char hist_indx;
  int rt_req_last_ttl; // last ttl value used
  // last few route discovery latencies
  // double               rt_length [MAX_HISTORY];
  // last few route lengths

  /*
   * a list of neighbors that are using this route.
   */
  aodv_ncache rt_nblist;
};


/*
  The Routing Table
*/

class aodv_rtable
{
public:
  aodv_rtable ()
  {
    LIST_INIT (&rthead);
  }

  aodv_rt_entry *head ()
  {
    return rthead.lh_first;
  }

  aodv_rt_entry *rt_add (nsaddr_t id);
  void rt_delete (nsaddr_t id);
  aodv_rt_entry *rt_lookup (nsaddr_t id);

private:
  LIST_HEAD (aodv_rthead, aodv_rt_entry) rthead;
};

//sandy
//service table - based on routing table
class aodv_svc_entry
{
  friend class AODV;
  friend class aodv_svc_table;
public:
    aodv_svc_entry ();
protected:
    u_int16_t port;
  char serv_string[32];
  nsaddr_t dest;
  double expire;
  u_int16_t hops;
    LIST_ENTRY (aodv_svc_entry) st_link;
};
class aodv_svc_table
{
  friend class aodv_svc_entry;
public:
    aodv_svc_table ()
  {
    LIST_INIT (&sthead);
  }
  aodv_svc_entry *head ()
  {
    return sthead.lh_first;
```

```
  }
  aodv_svc_entry *st_add (u_int16_t port, char *serv_string, nsaddr_t dest,
  double ltime, u_int16_t hops);
  void st_delete (aodv_svc_entry * st);
  aodv_svc_entry *st_lookup (nsaddr_t id);
  aodv_svc_entry *st_lookup (u_int16_t port);
  aodv_svc_entry *st_lookup (char *serv_string);
  void st_purge (nsaddr_t selfid);
private:
  LIST_HEAD (aodv_sthead, aodv_svc_entry) sthead;
};


#endif /* _aodv__rtable_h__ */
```

B.5 aodv_rtable.cc

```
/*Modified and extended by Sandeep Gupta, Indian Institute of Information
  Technology, Allahabad,
  India.
  May, 2003.
*/

/* Copyright (c) 2003 Indian Insitute of Information Technology, Allahabad.
  * All rights reserved.
  *
  * Redistribution and use in source and binary forms, with or without
  * modification, are permitted.
  *
  * The software is provided "as is" without express or implied warranty of any
  * kind arising out of the use of this software.
*/

/*
Copyright (c) 1997, 1998 Carnegie Mellon University.  All Rights
Reserved.

Permission to use, copy, modify, and distribute this
software and its documentation is hereby granted (including for
commercial or for-profit use), provided that both the copyright notice
and this permission notice appear in all copies of the software,
derivative works, or modified versions, and any portions thereof,
and that both notices appear in supporting documentation, and
that credit is given to Carnegie Mellon University in all publications
reporting on direct or indirect use of this code or its derivatives.

ALL CODE, SOFTWARE, PROTOCOLS, AND ARCHITECTURES DEVELOPED BY THE CMU
MONARCH PROJECT ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF
WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS
SOFTWARE OR OTHER INTELLECTUAL PROPERTY IN ITS ``AS IS'' CONDITION,
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY
BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE
OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE OR
INTELLECTUAL PROPERTY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

Carnegie Mellon encourages (but does not require) users of this
software or intellectual property to return any improvements or
extensions that they make, and to grant Carnegie Mellon the rights
to redistribute these changes without encumbrance.

The AODV code developed by the CMU/MONARCH group was optimized and tuned
by Samir Das and Mahesh Marina, University of Cincinnati. The work was
partially done in Sun Microsystems.
*/


#include <aodv/aodv_rtable.h>
//#include <cmu/aodv/aodv.h>

/*
  The Routing Table
*/

aodv_rt_entry::aodv_rt_entry ()
{
  int i;

  rt_req_timeout = 0.0;
  rt_req_cnt = 0;

  rt_dst = 0;
  rt_seqno = 0;
  rt_hops = rt_last_hop_count = INFINITY2;
  rt_nexthop = 0;
  LIST_INIT (&rt_pclist);
```

```
  rt_expire = 0.0;
  rt_flags = RTF_DOWN;

  /*
     rt_errors = 0;
     rt_error_time = 0.0;
   */


  for (i = 0; i < MAX_HISTORY; i++)
    {
      rt_disc_latency[i] = 0.0;
    }
  hist_indx = 0;
  rt_req_last_ttl = 0;

  LIST_INIT (&rt_nblist);

};


aodv_rt_entry::~aodv_rt_entry ()
{
  AODV_Neighbor *nb;

  while ((nb = rt_nblist.lh_first))
    {
      LIST_REMOVE (nb, nb_link);
      delete nb;
    }

  AODV_Precursor *pc;

  while ((pc = rt_pclist.lh_first))
    {
      LIST_REMOVE (pc, pc_link);
      delete pc;
    }

}


void
aodv_rt_entry::nb_insert (nsaddr_t id)
{
  AODV_Neighbor *nb = new AODV_Neighbor (id);

  assert (nb);
  nb->nb_expire = 0;
  LIST_INSERT_HEAD (&rt_nblist, nb, nb_link);

}


AODV_Neighbor *
aodv_rt_entry::nb_lookup (nsaddr_t id)
{
  AODV_Neighbor *nb = rt_nblist.lh_first;

  for (; nb; nb = nb->nb_link.le_next)
    {
      if (nb->nb_addr == id)
break;
    }
  return nb;

}


void
aodv_rt_entry::pc_insert (nsaddr_t id)
{
  if (pc_lookup (id) == NULL)
    {
      AODV_Precursor *pc = new AODV_Precursor (id);

      assert (pc);
```

```
      LIST_INSERT_HEAD (&rt_pclist, pc, pc_link);
    }
}


AODV_Precursor *
aodv_rt_entry::pc_lookup (nsaddr_t id)
{
  AODV_Precursor *pc = rt_pclist.lh_first;

  for (; pc; pc = pc->pc_link.le_next)
    {
      if (pc->pc_addr == id)
return pc;
    }
  return NULL;

}

void
aodv_rt_entry::pc_delete (nsaddr_t id)
{
  AODV_Precursor *pc = rt_pclist.lh_first;

  for (; pc; pc = pc->pc_link.le_next)
    {
      if (pc->pc_addr == id)
{
  LIST_REMOVE (pc, pc_link);
  delete pc;
  break;
}
    }

}

void
aodv_rt_entry::pc_delete (void)
{
  AODV_Precursor *pc;

  while ((pc = rt_pclist.lh_first))
    {
      LIST_REMOVE (pc, pc_link);
      delete pc;
    }
}

bool aodv_rt_entry::pc_empty (void)
{
  AODV_Precursor *
    pc;

  if ((pc = rt_pclist.lh_first))
    return false;
  else
    return true;
}

/*
  The Routing Table
*/

aodv_rt_entry *
aodv_rtable::rt_lookup (nsaddr_t id)
{
  aodv_rt_entry *rt = rthead.lh_first;

  for (; rt; rt = rt->rt_link.le_next)
    {
      if (rt->rt_dst == id)
break;
    }
  return rt;

}
```

```
void
aodv_rtable::rt_delete (nsaddr_t id)
{
  aodv_rt_entry *rt = rt_lookup (id);

  if (rt)
    {
      LIST_REMOVE (rt, rt_link);
      delete rt;
    }

}

aodv_rt_entry *
aodv_rtable::rt_add (nsaddr_t id)
{
  aodv_rt_entry *rt;

  assert (rt_lookup (id) == 0);
  rt = new aodv_rt_entry;
  assert (rt);
  rt->rt_dst = id;
  LIST_INSERT_HEAD (&rthead, rt, rt_link);
  return rt;
}

//sandy
aodv_svc_entry::aodv_svc_entry ()
{
  port = 0;
  bzero (serv_string, 32);
  dest = 0;
  expire = 0;
}

aodv_svc_entry *
aodv_svc_table::st_add (u_int16_t port, char *serv_string, nsaddr_t dest,
double ltime,u_int16_t hops)
{
  aodv_svc_entry *st = new aodv_svc_entry;
  st->port = port;
  strcpy (st->serv_string, serv_string);
  st->dest = dest;
  st->expire = CURRENT_TIME + ltime;
  st->hops = hops;

  LIST_INSERT_HEAD (&sthead, st, st_link);
  return st;
}

aodv_svc_entry *
aodv_svc_table::st_lookup (nsaddr_t id)
{
  aodv_svc_entry *st = sthead.lh_first;

  for (; st; st = st->st_link.le_next)
    {
      if (st->dest == id)
break;
    }
  return st;

}

aodv_svc_entry *
aodv_svc_table::st_lookup (u_int16_t port)
{
  aodv_svc_entry *st = sthead.lh_first;

  for (; st; st = st->st_link.le_next)
    {
      if (st->port == port)
break;
    }
  return st;

}
```

```
aodv_svc_entry *
aodv_svc_table::st_lookup (char *serv_string)
{
  aodv_svc_entry *st = sthead.lh_first;

  for (; st; st = st->st_link.le_next)
    {
      if (!strcmp (st->serv_string, serv_string))
break;
    }
  return st;

}

void
aodv_svc_table::st_delete (aodv_svc_entry * st)
{

  if (st)
    {
      LIST_REMOVE (st, st_link);
      delete st;
    }
}

void
aodv_svc_table::st_purge (nsaddr_t selfid)
{
  aodv_svc_entry *st = sthead.lh_first;

  for (; st; st = st->st_link.le_next)
    {
      //we donot want to delte a service entry offered by the host itself
      if (st->expire < CURRENT_TIME && st->dest != selfid)
st_delete (st);
    }
}
```

C Acronyms

- AODV: Ad-hoc On Demand Distance Vector protocol
- RREQ: Route Request
- RREP: Route Reply
- SREQ: Service Request extension
- SREP: Service Reply extension
- URL: Uniform Resource Locator