题目：无线自组网按需距离向量路由协议代码分析

小组成员：杨韬，叶上维，徐文博

分工：

目录

[1. 基本原理 5](#_Toc534019185)

[1.1 概述 5](#_Toc534019186)

[1.1.1 定义 5](#_Toc534019187)

[1.1.2特点 6](#_Toc534019188)

[1.2 AODV工作过程 6](#_Toc534019189)

[1.2.1 路由发现过程 7](#_Toc534019190)

[1.2.2 路由维护和错误控制 8](#_Toc534019191)

[1.2.3 拥塞控制 8](#_Toc534019192)

[1.3 AODV控制帧和机制介绍 8](#_Toc534019193)

[1.3.1 RREQ路由请求帧 9](#_Toc534019194)

[1.3.2 RREP路由应答帧 10](#_Toc534019195)

[1.3.3 RERR路由错误帧 11](#_Toc534019196)

[1.3.4 RREP\_ACK路由应答确认帧 11](#_Toc534019197)

[1.3.5 序列号机制 11](#_Toc534019198)

[1.3.6 前驱节点机制 12](#_Toc534019199)

[2. 代码介绍 12](#_Toc534019200)

[2.1 全局变量 13](#_Toc534019201)

[2.1.1 全局链表结构 13](#_Toc534019202)

[2.1.2 全局目的地链表结构 14](#_Toc534019203)

[2.1.3 全局定时器队列 14](#_Toc534019204)

[2.2 统计量 15](#_Toc534019205)

[2.3 SNMP变量 15](#_Toc534019206)

[3. 以太网接口 15](#_Toc534019207)

[3.1 main函数 15](#_Toc534019208)

[3.2 main函数文件内其他函数以及变量的定义： 18](#_Toc534019209)

[3.3 AODV\_socket.c 19](#_Toc534019210)

[3.3.1 \_\_cmsg\_nxthdr\_fix函数 19](#_Toc534019211)

[3.3.2 aodv\_socket\_init函数-套接字初始化函数 19](#_Toc534019212)

[3.3.3 aodv\_socket\_read函数-套接字读取 22](#_Toc534019213)

[3.3.4 aodv\_socket\_process\_packet函数-处理数据包函数 24](#_Toc534019214)

[3.3.5 recvAODVUUPacket函数-接收AODVUUPacket 25](#_Toc534019215)

[3.3.6 aodv\_socket\_send函数-套接字发送函数 26](#_Toc534019216)

[3.3.7 aodv\_socket\_new\_message函数 28](#_Toc534019217)

[3.3.8 aodv\_socket\_queue\_msg函数 29](#_Toc534019218)

[3.3.9 aodv\_socket\_cleanup函数 29](#_Toc534019219)

[3.4 AODV\_rreq.c 29](#_Toc534019220)

[3.4.1 rreq\_send函数-发送rreq消息 29](#_Toc534019221)

[3.4.2 rreq\_forward函数-转发rreq消息 29](#_Toc534019222)

[3.4.3 rreq\_process函数-处理rreq消息 30](#_Toc534019223)

[3.4.4 rreq\_route\_discovery函数-路由发现过程 34](#_Toc534019224)

[3.4.5 其他函数 36](#_Toc534019225)

[3.5 AODV\_rrep.c 36](#_Toc534019226)

[3.5.1 rrep\_ack\_process函数 37](#_Toc534019227)

[3.5.2 rrep\_send函数-发送rrep消息 38](#_Toc534019228)

[3.5.3 rrep\_forward函数-转发rrep消息 39](#_Toc534019229)

[3.5.4 rrep\_process函数-处理rrep消息 39](#_Toc534019230)

[3.5.5 其他函数 42](#_Toc534019231)

[3.6 aodv\_rerr.c 43](#_Toc534019232)

[3.6.1 rerr\_create函数-创建 43](#_Toc534019233)

[3.6.2 rerr\_process函数-处理 43](#_Toc534019234)

[3.6.3 其他函数 46](#_Toc534019235)

[3.7 Time\_queue.c 46](#_Toc534019236)

[3.7.1 NS\_CLASS timer\_init函数-记时器初始化函数 46](#_Toc534019237)

[3.7.2 NS\_CLASS timer\_timeout函数-计时器超时，调用此函数进行清理工作 47](#_Toc534019238)

[3.7.3 NS\_CLASS timer\_add函数-在计时器队列中添加计时器函数 48](#_Toc534019239)

[3.7.4 NS\_CLASS timer\_remove函数-计时器队列中移除计时器 49](#_Toc534019240)

[3.7.5 timer\_left函数-计算计时器剩余时间 49](#_Toc534019241)

[3.7.6 timer\_age\_queue函数-计算计时器队列的生存时间 50](#_Toc534019242)

[3.8 aodv\_hello.c 51](#_Toc534019243)

[3.8.1 hello\_start()函数-启动hello消息 51](#_Toc534019244)

[3.8.2 hello\_send()函数-发送hello消息 51](#_Toc534019245)

[3.8.3 hello\_process()函数-处理hello消息 54](#_Toc534019246)

[3.8.4 其他函数 55](#_Toc534019247)

[3.9 Nl.c 56](#_Toc534019248)

[3.9.1 Nl\_init函数-初始化函数 56](#_Toc534019249)

[3.9.2 Nl\_kaodv\_callback函数-根据aodv套接字中收到的不同命令，执行不同的回调函数 57](#_Toc534019250)

[3.9.3 Nl\_rt\_callback函数-根据路由套接字中收到的不同命令，执行不同的回调函数 63](#_Toc534019251)

[3.9.4 Nl\_send函数-发送数据包给内核模块 65](#_Toc534019252)

[3.9.5 Nl\_kern\_route函数-操作内核路由表，包括增加、修改、删除路由表条目等操作 66](#_Toc534019253)

[3.9.6 Nl\_send\_add\_route\_msg函数-向内核发送一个添加一个路由信息的消息，包括目的地地址、下一跳，生存期，路由状态标志，网络接口等 67](#_Toc534019254)

[3.10 Aodv\_neighbor.c 67](#_Toc534019255)

[**3.10.1更新活跃的邻居节点（更新来自非HELLO AODV控制消息的邻居）（void NS\_CLASS neighbor\_add）** 67](#_Toc534019256)

[3.10.2邻居节点链路中断情况（void NS\_CLASS neighbor\_link\_break） 68](#_Toc534019257)

[3.11 Aodv\_timeout.c 69](#_Toc534019258)

[3.11.1定义定时器到期时调用的超期函数 70](#_Toc534019259)

[3.11.2.定义本地修复超时处理函数 70](#_Toc534019260)

[3.11.3.设置路由过期超时函数 71](#_Toc534019261)

[3.11.4.路由删除函数设置 71](#_Toc534019262)

[3.11.5.HELLO消息超时函数 71](#_Toc534019263)

[3.11.6响应超时函数 71](#_Toc534019264)

[3.12 kaodv-netlink.h 72](#_Toc534019265)

[3.13 kaodv-netlink.c 73](#_Toc534019266)

[3.13.1kaodv\_netlink\_build\_msg-创建sk\_buff结构体 74](#_Toc534019267)

[3.13.2 kaodv\_netlink\_receive\_peer-内核收到消息后的处理房产税的方式 76](#_Toc534019268)

[4. 4 AODV仿真 77](#_Toc534019269)

[4.1仿真思路： 77](#_Toc534019270)

[4.2 获取trace文件 77](#_Toc534019271)

[4.2.1生成仿真环境 77](#_Toc534019272)

[4.2.2 编写aodv.tcl脚本 78](#_Toc534019273)

[4.2.3 执行aodv.tcl脚本 78](#_Toc534019274)

[4.3.2 路由发起频率 80](#_Toc534019275)

[4.3.3 路由开销 81](#_Toc534019276)

[4.3.4 平均时延 82](#_Toc534019277)

[5. 结束语 83](#_Toc534019278)

# 基本原理

## 概述

AODV协议(Ad hoc on-demand distance vector routing)是应用于无线自组织网络中进行路由选择的路由协议,它能够实现单播和多播路由。该协议是自组织网络中按需生成路由方式的典型协议。用于特定网络中的可移动节点。它能在动态变化的点对点网络中确定一条到目的地的路由，并且具有接入速度快，计算量小，内存占用低，网络负荷轻等特点。它采用目的序列号来确保在任何时候都不会出现回环（在路由控制信息出现异常的时候也是如此），避免了传统的距离向量协议中会出现的很多问题。

### 1.1.1 定义

AVOD算法旨在多个移动节点中建立和维护一个动态的、自启动的、多跳路由的网络。它是的移动节点可以快速获得新目的节点的路由，而每一个节点仅需维护信号范围内节点的路由，更远的节点路由信息不必维护。

AODV定义了三种消息种类，分别是：路由请求（RREQ）、路由回复（RREP）、路由错误（RERR）。这些消息通过UDP和通常的IP协议来接收。消息不会被盲目不转发，但AODV允许特殊操作的得到广泛分布（e.gRREQ），甚至散布到整个无线自组织网络。这些RREQ的散布范围由IP头文件中的TTL来确定。

在广播RREQ消息之前，发起节点将消息的缓存一段时间。这个时间由“PATH\_DISCOVERY\_TIME”来决定。当节点从邻居节点收到具有同样RREQID和OriginatorIPaddress的RREQ消息时，他江晖认为这是一个发回来的包而将其丢弃。当某节点连接一个新节点时，广播RREQ尝试找到到目的节点的路由。在两种情况下，路由可以被找到：首先，RREQ消息到达目的节点，路由被找到。其次，如果RREQ消息到了一个中间节点，这个中间节点拥有到目的节点足够新（freshness）的路由（freshness的路由指的是有一条到目的节点的正确路由，且拥有一个较大的序列号，该序列号不得小于RREQ的序列号）。当朝发起RREQ的节点单播一个RREP消息后，该路由建立。所有接收到请求的结点都会缓存一条回到发起节点的路由，因此RREP可以通过单播从目的节点返回到发起节点，或则从一个能找到目的节点的中间节点返回到发起节点。

当活动路由表有一条连接断开时，一条RERR消息将被用来通知其他节点发生连接断裂。RERR消息指出不能到达的目的节点（或者目的子网）。为了实现这种报告机制，接地啊你将会维护一张“先驱表”，其中的信息可以轻易从回传的RREP过程中获得。先驱表中包含邻居节点IP，这些邻居节点会将它用作到达目的地的下一跳节点。先去表

AODV维护了当一个节点需要给网络中的其他节点传送信息时，如果没有到达目标节点的路由，则必须先以多播的形式发出RREQ(路由请求)报文。RREQ报文中记录着发起节点和目标节点的网络层地址，邻近节点收到RREQ，首先判断目标节点是否为自己。如果是，则向发起节点发送RREP(路由回应)；如果不是，则首先在路由表中查找是否有到达目标节点的路由，如果有，则向源节点单播RREP，否则继续转发RREQ进行查找。

在网络资源充分的情况下，AODV协议可以通过定期广播hello报文来维护路由，一旦发现某一个链路断开，节点就发送ERROR报文通知那些因链路断开而不可达的节点删除相应的记录或者对已存在的路由进行修复。

### 1.1.2特点

1. AODV协议采用的基本路由算法为距离向量算法，但有所改进。
2. AODV协议是按需路由协议。也就是说，只有在主机需要的时候才会激发出路由寻找过程。
3. AODV采用UDP封装，属于应用层协议。每次寻找路由时都要触发应用层协议，增加了实现的复杂度。
4. IEEE 802．15．4规定的帧大小为127字节，MAC头部及尾部校验最多使用了25字节，IPv6头部与UDP头部占用 48字节，只剩下54字节的空余。
5. LR—WPAN中，拓扑结构相对简单，网络的规模相对较小，节点的位置不固定，对它的设计首先要考虑的因素是简单、节能等问题。AODV的路由框架和报文设计并没有考虑这些问题。

AODV是一个路由协议，它的工作是管理路由表。AODV的每一个路由表将包含以下内容：

每个AODV协议的结点都要维护一张路由表，表的各字段如下：

1. 目的节点IP地址
2. 目的节点序列号（Sequence Number）
3. 目的节点序列号是否有效标志位
4. 下一跳节点的IP地址
5. 本节点到达目的节点需要的跳数
6. 前驱节点列表（precursor list）
7. 生存时间（路由失效或应当删除的时间）
8. 网络层接口
9. 其他的状态和路由标志位（e.g有效、无效、可修复、正在修复）

路由表的特征如下：

1. 路由表每项只记录下一跳路由信息，而不是整条路由信息，简化了路由表的建立和维护 。
2. 节点将相邻节点的连接状态保存到活动路由表中。
3. AODV在每一个路由表上使用了目的序列号。 源节点和目的节点都维护各自的序列号 。
4. 当一条链路断裂或失效导致一个节点不可达，路由表将通过对序列号的操作和标注路由表表项错误来使此条路由失效。

## AODV工作过程

aodv协议包括路由发现，路由维护，路由错误控制，拥塞控制4种工作过程。它通过路由发现添加新的路由表项，通过路由维护保证出入口的通畅，路由错误控制来剔除失效的路由项，拥塞控制降低网络负载。

### 路由发现过程

AODV协议是按需平面距离向量路由协议，所以只有当上层有需要发数据且路由表项不存在的时候，才触发aodv的路由发现功能。aodv通过rreq帧和rrep帧实现本功能。

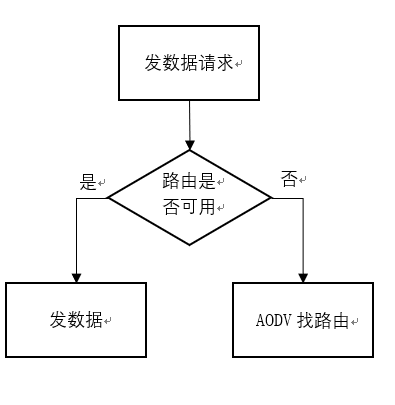


图 1 发现功能触发

首先，作为起始节点向邻居节点发送RREQ帧。邻居节点收到后，也向它的邻居节点转发RREQ帧。最后目标节点也会收到RREQ帧。

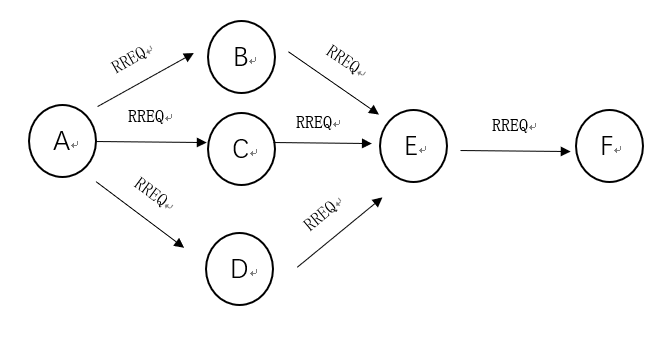


图 2 RREQ帧转发过程

在RREQ帧传播的过程中，各个中间节点在收到帧后更新到源端的路由。对于收到多个源端到一个目的端的RREQ帧的情况，取最早到达的更新到源端的路由表项。收到同一个源端到多个目的端的RREQ帧，取最新的更新到源端的路由表项。

最终目的节点收到RREQ帧后，也更新路由表项，同时查询路由表，向源端回复一个RREP帧。中间节点收到后更新路由表，查询路由表，向源端单播RREP帧。

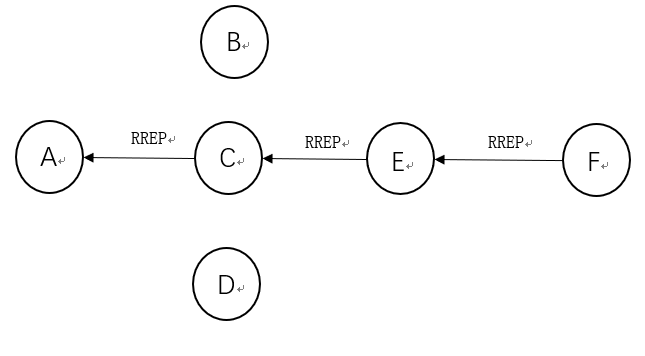


图 3 单播RREP过程

最后源端收到RREP回复帧后，更新路由表项，数据包得以发送。

还有一种情况是，当RREQ帧中相关字段允许的话，中间节点收到RREQ后，查找自己的路由表，如果发现有到达目的地的路由表项，则回复一个RREP帧。

### 路由维护和错误控制

AODV通过发送HELLO帧来实现路由的维护功能，保证了节点各个出口的有效性。

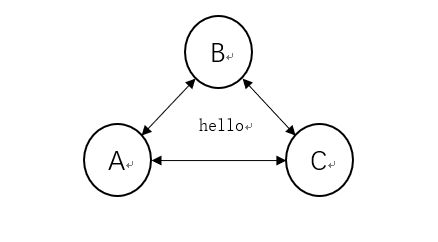


图 4 节点向邻居发hello帧

HELLO帧本质上是TTL为1的RREP帧。当节点收到RREP帧后更新对应的计时器，重

新计时。节点发现自己路由项中某个计时器超时，将该链路设置为无效，同时向相关路由表项中的前期节点发送RERR错误控制帧。

各节点收到RERR后根据帧内容修改路由表，同时把RERR转发。

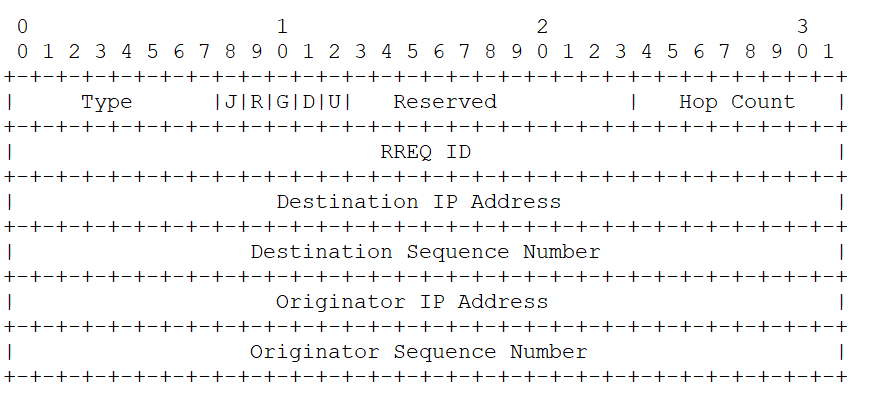
### 拥塞控制

如果在规定时间内，发送RREQ请求帧的节点没有收到对应的RREP帧，那么节点可以再次发送RREQ请求帧。再次发送RREQ帧时，节点对于RREP帧的等待时间将会翻倍，这使得网络的负担得以减少。在尝试了设定次数后，节点就在路由表中标志该目的地不可达，然后构造消息发送给应用层。

## AODV控制帧和机制介绍

AODV控制帧主要包括RREQ路由请求帧、RREP路由应答帧、RERR路由错误帧、RREP\_ACK路由应答确认帧、HELLO活跃路由链路检测帧。其中HELLO帧其实是TTL=1的RREP路由请求帧。同时介绍AODV路由的序列号和前驱节点机制。

### RREQ路由请求帧



具体每个字段含义如下：

Type 1（这个字段填写帧的类型，1代表RREQ，2:RREP,3:RERR）

J 加入标志; 为多播保留

R 修复标志，为多播保留

G Gratuitous RREP flag; 决定了是否向目标节点地址域指定的节点发送一个免费路由回复消息

D Destination only flag; 决定了是否只允许目标节点回复本条路由请求帧

U Unknown sequence number; 指示目标节点序列号未知.

Reserved 置为0，接受时忽略此字段

Hop Count 从源节点到包处理节点的跳数

RREQ ID 唯一标识一个从源IP节点发出的RREQ请求帧

Destination IP Address

目的地IP地址

Destination Sequence Number

源IP节点在其路由表能找到的最大的目的IP的序列号

Originator IP Address

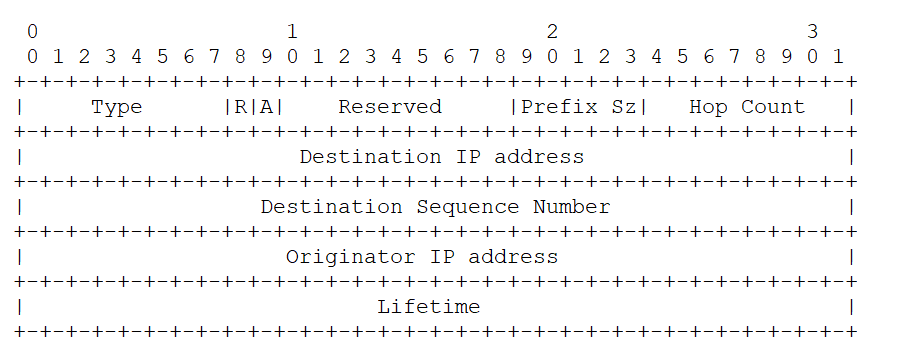
源IP地址，发起路由请求的节点IP地址

Originator Sequence Number

源IP地址的序列号

在源节点发送RREQ路由请求时，它会将种类，和关键字填好，设置一个RREQ ID来唯一标识这个包。从路由表中寻找目的IP最大的序列号填入，自己的序列号填入。当中间节点收到RREQ帧时，通过读取RREQ ID就可以知道之前有没有收到过这次路由请求。然后把HOP Count加1，转发出去。直到目的节点收到这个包。

### RREP路由应答帧



相同的字段就不再重复介绍了，介绍独有的：

R Repair flag; 修复标志，用于广播

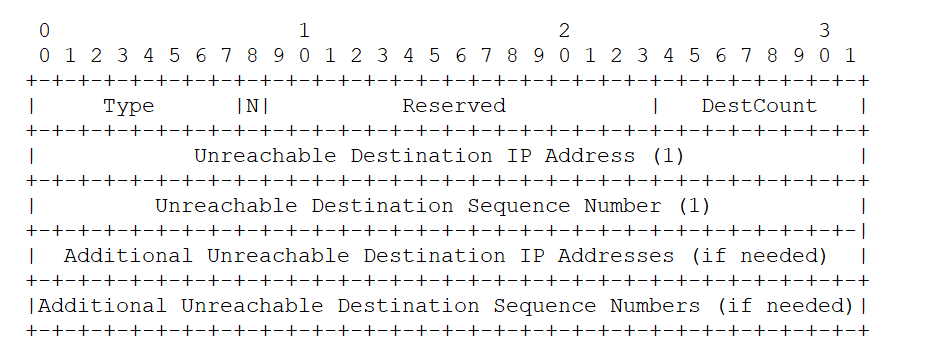
A Acknowledgment required;需要确认位，当发送RREP消息的链路可能不可靠或单向时，置A位，这决定了收到该RREP的节点是否需要回复RREP\_ACK包.

Prefix Size 前缀长度，字段大小为5bit。如果不是0，那么下一跳节点可以作为任何具有相同长度路由前缀的节点被请求时的目的节点.。

Lifetime 路由生命时间，这段时间内，收到这条路由的节点会认为这条路由是有效的

请注意，前缀大小允许子网路由器为由路由前缀定义的子网中的每个主机提供路由，路由前缀由子网路由器的IP地址和前缀大小决定。为了利用这个特性，子网路由器必须保证所有共享指定子网前缀的主机都能访问。详见第7节。当前缀大小不为零时，必须保留与子网路由有关的任何路由信息（和前体数据），而不是该子网上的单个目标IP地址。

### RERR路由错误帧



N No delete flag; 当上一个节点已经对这条连接进行本地修复了则设置这个标志位，这样上游的节点就不用删除这条路由了

DestCount 不可达节点数，至少为1；

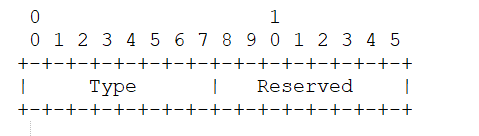
Unreachable Destination IP Address

由于连接断开而不可达的节点

Unreachable Destination Sequence Number

不可达节点的序列号

### RREP\_ACK路由应答确认帧



当节点收到的RREP回复帧中A位置为有效时，节点需要回复RREP\_ACK帧，这通常是为了避免存在单向链路时，路由发现周期无法完成的危险。

### 序列号机制

为了保证节点的路由表上的路由表项是最新的，所有节点都只使用收到的包含待更新路由节点最大序列号的包来更新自己的路由表。每个节点都有自己的序列号，每当节点发一次包这个序列号就+1。每个节点记录下收到节点包里最大的序列号，存放到相应的路由表项中。所以当你收到包的序列号大于你路由对应表项里的序列号，你就知道你收到了更新的包，然后更新路由表，不然使用改包更新路由表。

### 前驱节点机制

当链路出现中断时，相关节点需要更新自己的路由。如何确定哪些节点受到了影响呢？这就通过前驱节点得知。当节点直连链路出现中断，它只会向前驱节点发送RERR路由错误帧，收到RERR错误帧的节点更新自己的路由表，同时接着向其他受影响的前驱节点发送RERR错误帧，直到所有受影响的节点都收到RERR错误帧。

每个节点会为路由项维护一个前驱节点列表。当节点收到RREP的时候，会将到RREP包源节点下一跳的节点作为前驱节点添加到本节点到RREP包目的节点的路由项中，同时将到达RREP包目的节点的下一跳节点作为前驱节点加入到本节点到RREP包源节点的路由项中。然后收到RERR路由错误帧时，只向前驱节点转发，就完成了路由信息的更新。

见下图：

RREP

A路由发现E，当C收到回复的RREP时，就把B,D添加为两条路由的前驱节点。

如果收到RERR关于不可达节点A的，那么RERR就会转发给D

C路由表

目的节点 下一跳节点 前驱节点

A B D

E D B

# 代码介绍

该协议实现代码的根目录下共包含17个c文件和17个头文件里；一个记录更改日志的Changelog文件；一个GNU通用公共授权文件GPL；一个用于编译的Makefile文件；一个用于简要介绍AODV和使用方法的README文件；一个lnx文件夹，里面放了AODV协议用于linux 内核实现的代码，共6个c文件和7个头文件；一个patch文件夹，其中有8个补丁文件。

|  |  |
| --- | --- |
| 文件 | 说明 |
| Params.h | 各种参数的定义 |
| routing\_table.h | 路由表定义 |
| routing\_table.c | 路由表操作 |
| defs.h | 宏定义 |
| list.h | 链表结构定义 |
| list.c | 链表操作 |
| timer\_queue.h | 定时器队列定义 |
| timer\_queue.c | 定时器操作 |
| debug.h | 日志操作、转换成string、打印路由表 |
| debug.c |
| Endian.c | 补充某些缺乏〈endian.h〉的系统 |
| Aodv\_hello.h | hello消息的定义 |
| Aodv\_hello.c | hello消息的发送、处理等操作 |
| Aodv\_rerr.h | RERR消息的定义 |
| Aodv\_rerr.c | RERR消息的操作 |
| Aodv\_rrep.h | RREP消息的定义 |
| Aodv\_rrep.c | RREP消息的操作 |
| Aodv\_rreq.h | RREQ消息的定义 |
| Aodv\_rreq.c | RREQ消息的操作 |
| Aodv\_socket.h | AODV套接字定义 |
| Aodv\_socket.c | AODV套接字管理 |
| Aodv\_timeout.h | AODV超时处理 |
| Aodv\_timeout.c |
| Aodv\_neighbor.h | 添加和断开邻居 |
| Aodv\_neighbor.c |

## 全局变量

### 全局链表结构

全局维护的一个链表结构，从Linux内核列表实现中获得灵感而仿造的简单链接列表，具体实现在list.c文件中。广泛用于各个功能，比如邻居节点列表，计时器列表，前驱节点列表等，方便实例化和操作列表。

|  |  |
| --- | --- |
| 名称 | 功能 |
| typedef struct list\_t | 列表结构体，下面的函数是对列表的操作 |
| static inline int listelm\_detach(list\_t \* prev, list\_t \* next) | 删除某位置上的节点 |
| static inline int listelm\_add(list\_t \* le, list\_t \* prev, list\_t \* next) | 在指定位置添加节点 |
| int list\_add(list\_t \* head, list\_t \* le) | 在头部添加一个节点 |
| int list\_add\_tail(list\_t \* head, list\_t \* le) | 在末尾添加一个节点 |
| int list\_detach(list\_t \* le) | 删除指定节点 |

### 全局目的地链表结构

在路由发现功能中，RREQ发出后不会立即获得回复，而此时可能需要发出另一个节点RREQ请求，所以需要一个链表结构来维护RREQ请求的目的地节点。具体实现在Seek\_list.h和Seek\_list.c文件中。

|  |  |
| --- | --- |
| 名称 | 功能 |
| seek\_list\_t \*NS\_CLASS seek\_list\_insert  (struct in\_addr dest\_addr, u\_int32\_t dest\_seqno, int ttl, u\_int8\_t flags, struct ip\_data \*ipd) | 在链表中插入新节点，即增加了一个需要寻找的目的地 |
| int NS\_CLASS seek\_list\_remove(seek\_list\_t \* entry) | 从列表中删除一个需要寻找的目的地节点 |
| seek\_list\_t \*NS\_CLASS seek\_list\_find(struct in\_addr dest\_addr) | 从列表中找出指定地址的节点 |
| void NS\_CLASS seek\_list\_print() | 打印出每个待寻找目的节点的地址，序列号等信息 |

### 全局定时器队列

AODV很多地方需要使用到定时器，比如说路由表项的维护，邻居活跃情况，还有路由修复等，需要定时器在需要的时候触发超时事件。

|  |  |
| --- | --- |
| 名称 | 功能 |
| int NS\_CLASS timer\_init(struct timer \*t, timeout\_func\_t f, void \*data) | 定时器初始化 |
| void NS\_CLASS timer\_timeout  (struct timeval \*now) | 定时器超时时进行清理工作 |
| NS\_STATIC void NS\_CLASS timer\_add  (struct timer \*t) | 在定时器队列中加一个定时器 |
| int NS\_CLASS timer\_remove(struct timer \*t) | 从定时器队列中移除一个定时器 |
| int NS\_CLASS timer\_timeout\_now  (struct timer \*t) | 手动让一个定时器超时，以触发超时事件 |
| void NS\_CLASS timer\_set\_timeout(struct timer \*t, long msec) | 给指定定时器增加一定的时间 |
| long timer\_left(struct timer \*t) | 计算指定定时器剩余时间 |
| struct timeval \*NS\_CLASS timer\_age\_queue() | 计算定时器队列的生存时间 |
| void NS\_CLASS printTQ(list\_t \* l) | 打印出每个定时器的信息 |

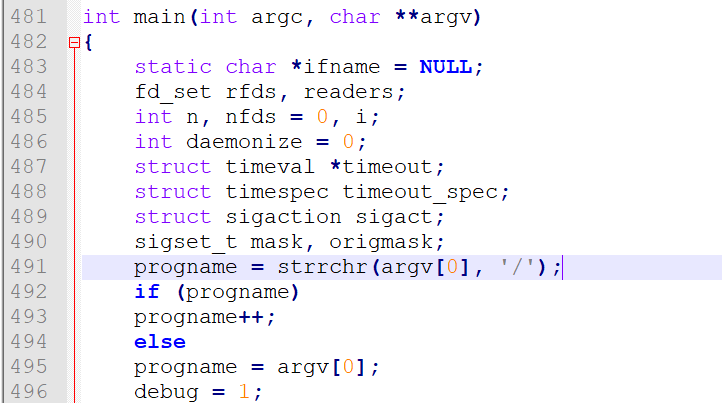
## 统计量

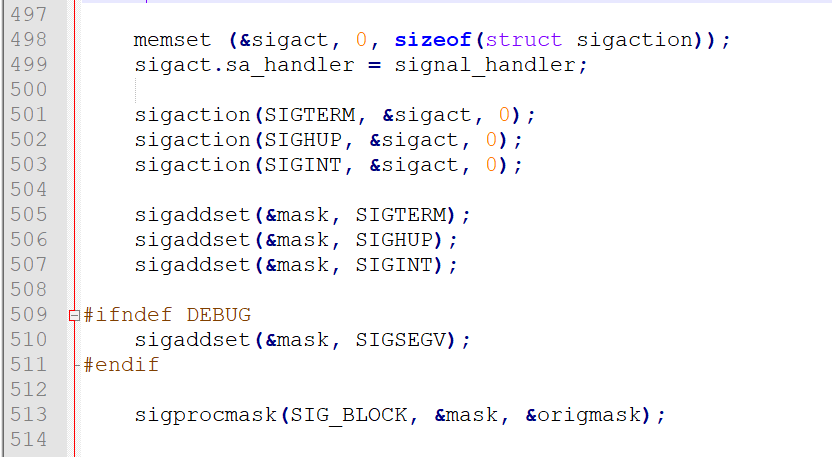
## SNMP变量

# 以太网接口

## main函数

main函数主要功能是对于各个变量的初始化，包括应用到的套接字，全局链表，路由表，信号等。读取命令行参数以对相关变量进行设置，使得协议执行不同的动作。最终通过pselect轮询函数来实现节点的各个功能。





481-496 声明各变量，包括文件描述符集，计时器，信号量，读取文件名等

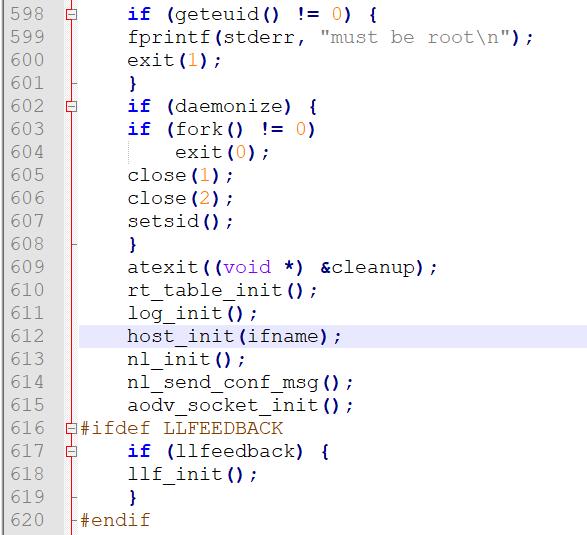
497-514 先是通过memset函数来清空结构体的内存，然后对信号结构体进行相关设置，最后将信号结构体阻塞住，将来在pselect函数中处理它。

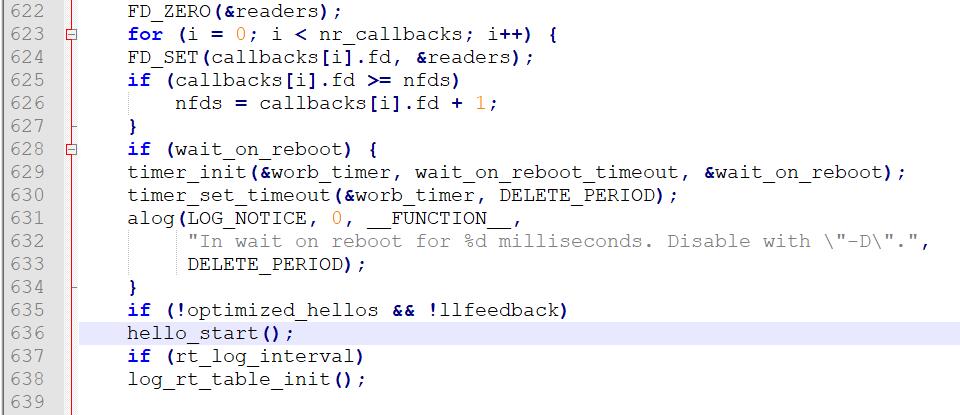




515-596 这是一个永真循环，它读取用户输入的命令行参数，然后根据输入的命令执行AODV不同的工作分支。具体操作，如下表所示：

|  |  |  |
| --- | --- | --- |
| 命令行参数 | 全称 | 说明 |
| -d | daemon | 开启守护进程模式，离开控制台窗口 |
| -f | llfeedback | 开启链路层反馈 |
| -g | force-gratuitous | 在每个传输的消息上设置gratuitous标志 |
| -i | interface | 后接端口，表示绑定到某端口；如没有，则自动分配 |
| -j | hello-jitter | 启用hello-jitter服务，默认也是启用的 |
| -l | log | 显示日志 |
| -n | n-hello | 将某节点设置成活跃邻居所需要接收到HELLO帧数 |
| -o | opt-hellos | 设置只在转发消息时才发送hello数据包 |
| -q | quality-threshold | 设置消息质量的最小阙值 |
| -r | log-rt-table | 每个一段时间记录一次路由表 |
| -u | unidir-hack | 开启检测服务 |
| -w | gateway-mode | 开启网管支持 |
| -x | no-expanding-ring | 禁用RREQ的拓展环搜索法 |
| -L | local-repair | 开启本地修复 |
| -D | no-worb | 禁用开启延迟的等待 |
| -R | ratelimit | 开启RREP和RREQ的速率限制 |
| -V | version | 输出版本信息 |
| -: | exit | 退出 |





598-601  检查AODV是否在root权限下运行，否则退出

602-608  检查变量daemonize，若不等于0，检测当前是否是父进程，是父进程则退出，不是父进程，就把输入，输出，错误流都关闭了，并且使当前进程成为新的会话组长和新的进程组长，并与原来的登录会话和进程组脱离。

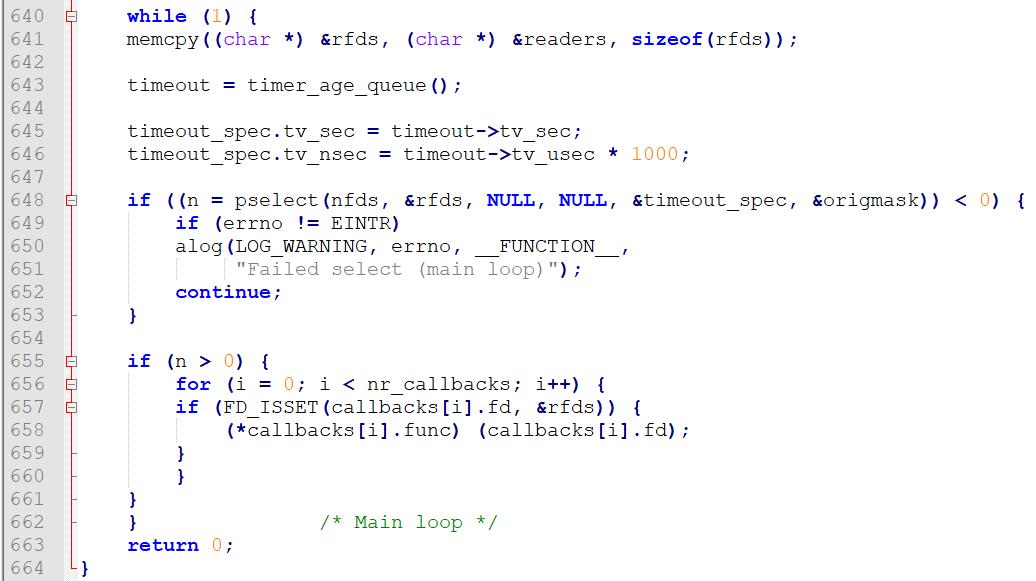
609-619  调用各初始化函数，初始化路由表，日志，各种套接字。

622-627 把需要监听的套接字描述符加入到文件描述符集中去，为以后pselect使用作准备

628-634 如果wait\_on\_reboot不为零，设置重启定时器，超出一定时间进行重启

635-637 启动HELLO帧，准备发第一个

637-369 初始化日志路由表



641-641 复制文件描述符集

643-646 设置定时器

648-653 pselect轮询函数，当文件描述符集中出现描述符对应套接字可读的时候，函数返回可读的个数n。

655-661 如果套接字可读数大于0，则通过FD\_ISSET遍历文件描述符集，判断具体是哪个可读，然后执行对应的回调函数。回调函数通过attach\_callback\_func函数来指定。

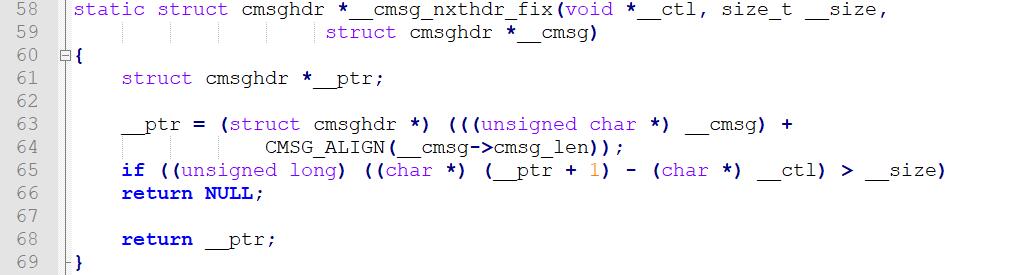
## main函数文件内其他函数以及变量的定义：

在这里以表格的形式简单说明main.c中剩下的函数功能和变量

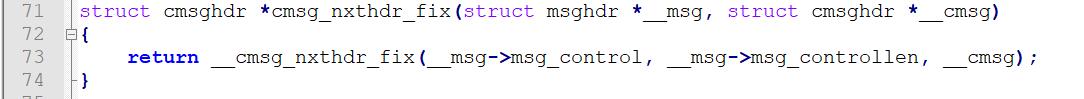
|  |  |
| --- | --- |
| 名称 | 说明 |
| voidusage(intstatus) | 打印命令行参数和说明 |
| voidcleanup() | 里面调用了各个部件的cleanup函数，实现了清空套接字、路由表等信息的功能 |
| struct callback | Callback结构体，成员变量包含一个int型的文件描述符和一个回调函数型的函数 |
| int attach\_callback\_func(int fd, callback\_func\_t func) | 增加一个callback元素 ，设置它的套接字描述符和对应的函数 |
| int find\_default\_gw(void) | 寻找节点默认网关 |
| struct sockaddr\_in \*get\_if\_info(char \*ifname, int type) | 根据套接字接口名字和其类型获取详细信息 |
| void host\_init(char \*ifname) | 初始化某个端口 |
| void load\_modules(char \*ifname) | 装载内核模块 |
| void remove\_modules(void) | 删除某个模块 |
| int set\_kernel\_options() | 设置内核选项 |
| void signal\_handler(int type) | 信号处理函数，根据不同类型的信号作不同的操作 |

## AODV\_socket.c

### \_\_cmsg\_nxthdr\_fix函数



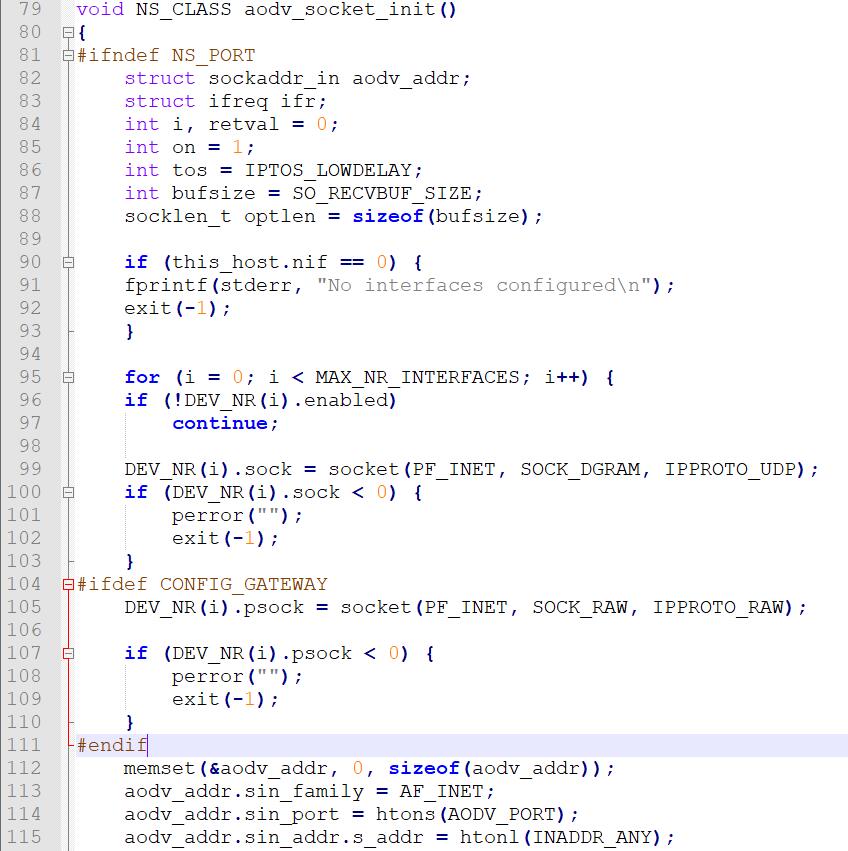
58-69首先输入的参数创建一个类型为cmsghdr的结构变量ptr，然后检查\_ptr+1减去\_\_ctl是否大于\_size,如果条件为真，则返回空，否则返回创建好的结构。该函数重写了libc里的CMSG\_NXTHDR函数。



71-74 返回调用\_\_cmsg\_nxthdr\_fix函数的结果，即构建好的结构体对象

### aodv\_socket\_init函数-套接字初始化函数

该函数主要实现了对应aodv套接字的初始化。可以清晰的看到，该套接字是用UDP协议进行传输的。可以看到函数末尾调用了attach\_callback\_func函数，将自己的套接字描述符和aodv处理函数作为参数传了进去。说明这个套接字是用来接收aodv控制帧的，当套接字可读的时候，调用了aodv\_socket\_read函数来处理接收到的包。

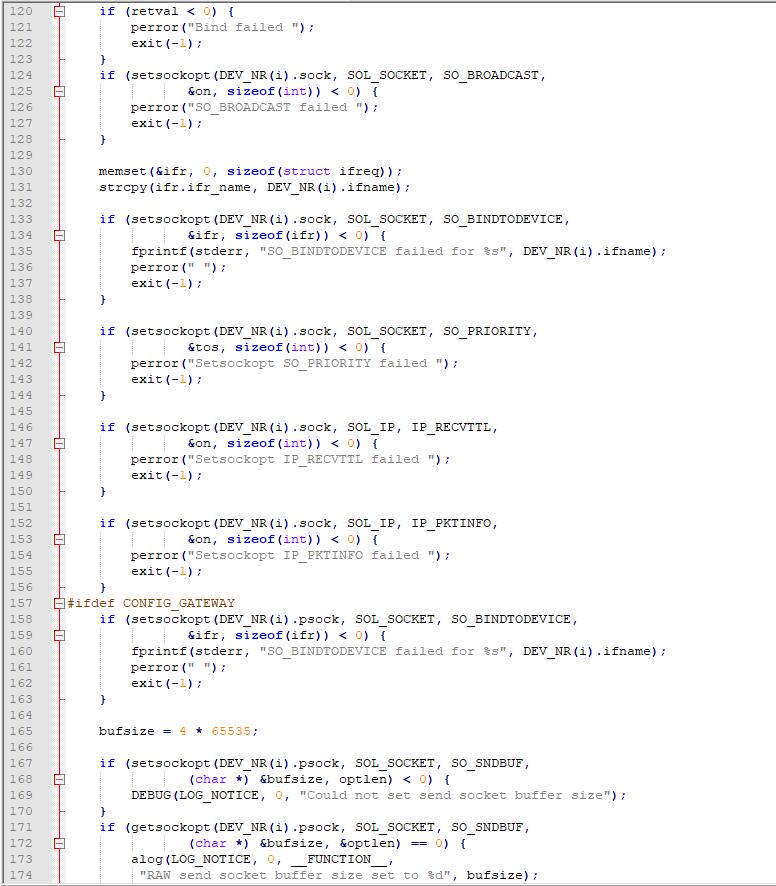


79-88 声明UDP套接字所需要的变量

90-93 如果没有可用接口的话之间退出

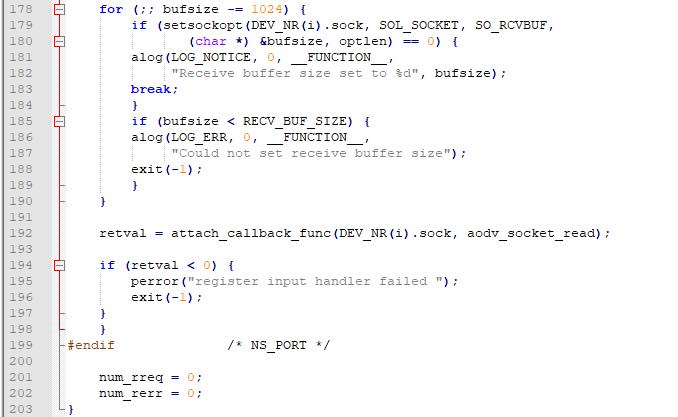
95-97 这个循环将在规定值范围内，为所有可用的接口创建套接字连接。

99-115 创建并检测生成的UDP套接字和与内核沟通的原始套接字，设置aodv地址参数





120-176 设置套接字选项，如果其中一个设定失败则显示错误，并返回



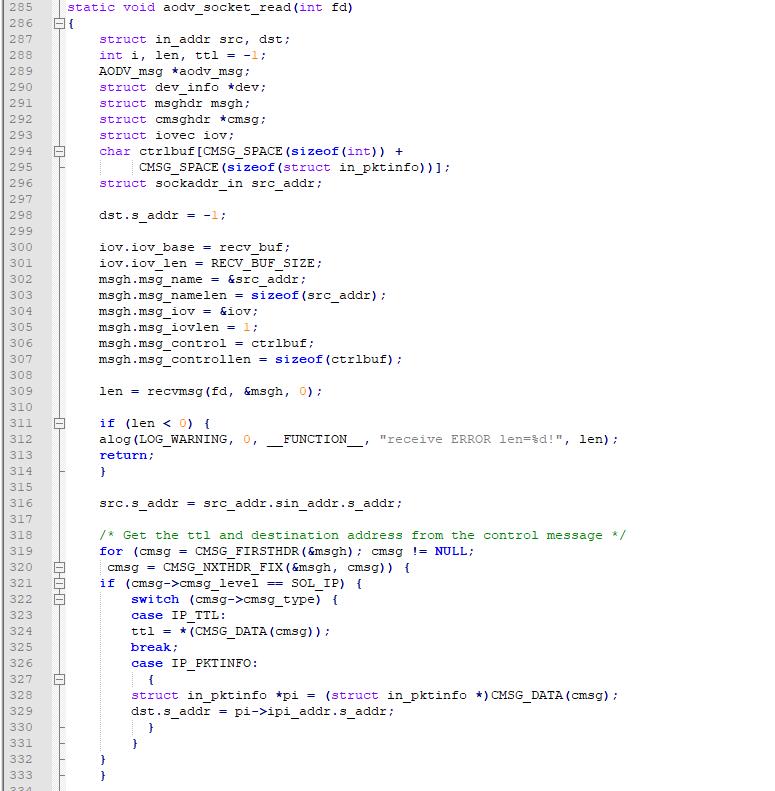
178-190 设置收控制帧的缓冲区大小，失败则显示消息并返回

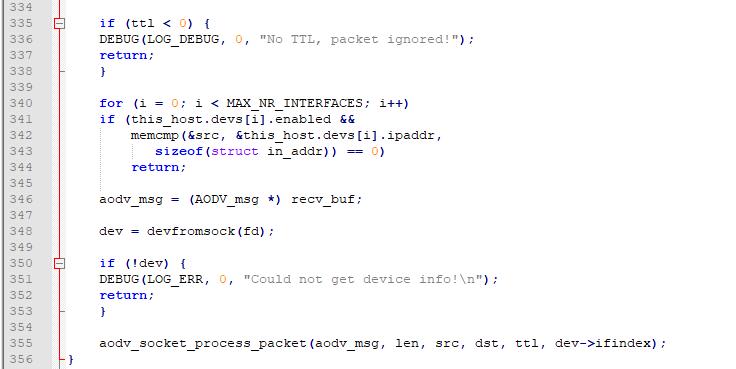
192-192 把创建好的套接字挂载到主函数的callback结构体当中

117-123 绑定套接字和AODV端口号

### aodv\_socket\_read函数-套接字读取

该函数被设定为callback结构体中某成员的回调函数，当aodv套接字可读时将会调用这个函数对收到的包进行处理。这个函数对接收到的aodv控制帧包进行过滤操作，丢弃长度小于0的和本地生成的数据包。因为发送和接受用同一个套接字进行传输，当我们需要发送帧时，把信息写入缓冲区中，套接字同样可读，会调用这个函数，所以需要判断。





287-307 声明各消息变量，以待使用

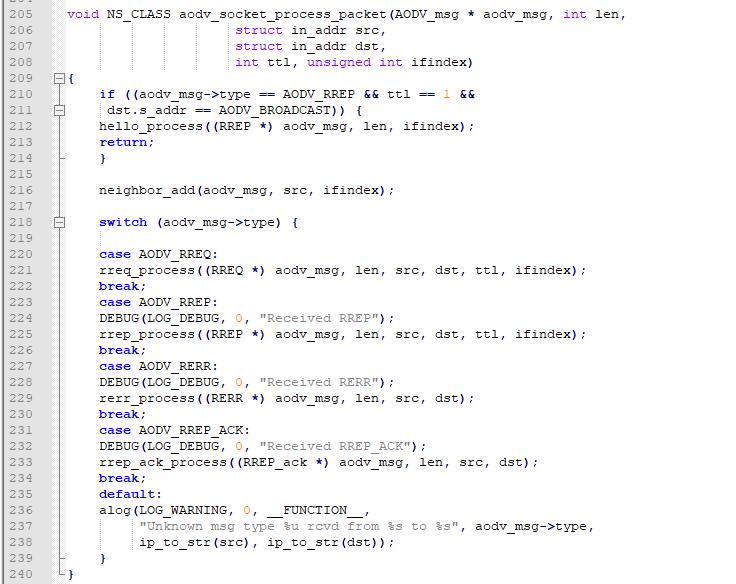
309-314 收取套接字接收到的信息，如果长度小于0，则丢弃

319-344 获取控制帧的ttl和目的地址，判断如果ttl<0或者是本节点生成的数据包则忽略掉。

346-355 提取相应消息，把包传给aodv\_socket\_process\_packet函数进行进一步处理

### aodv\_socket\_process\_packet函数-处理数据包函数

控制帧在经过aodv\_socket\_read函数初步筛选过后递交给本函数进行进一步处理，本函数主要实现的是一个分发处理功能。收到包后判断其类型，递交给对应的处理函数进行工作。

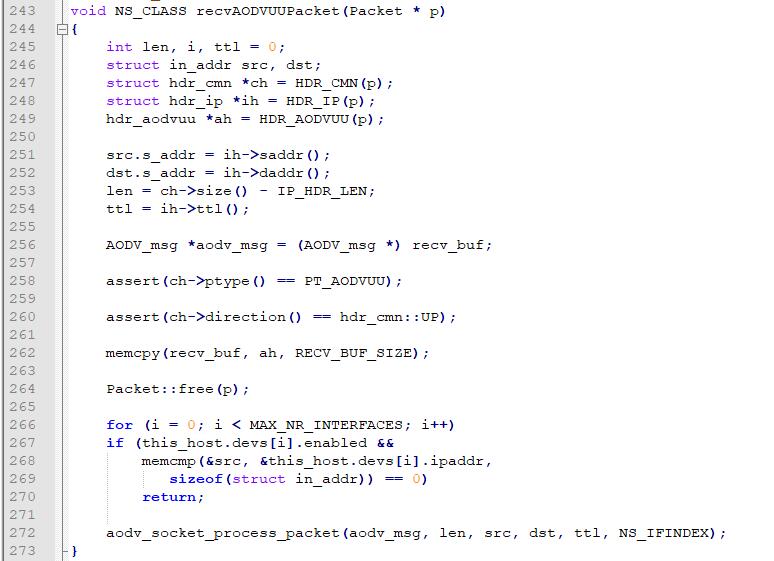


210-214 如果包是一个HELLO帧，则把该包传给hello\_process函数进行处理

226-226 根据aodv消息，src和ifindex参数添加邻居节点

218-239 提取包类型，将包递交给对应的函数进行处理

### recvAODVUUPacket函数-接收AODVUUPacket

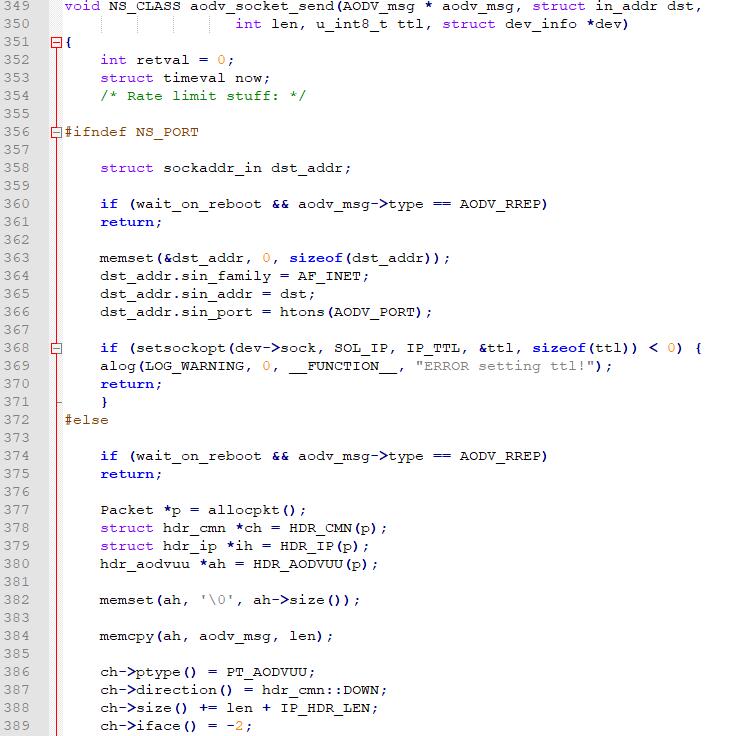


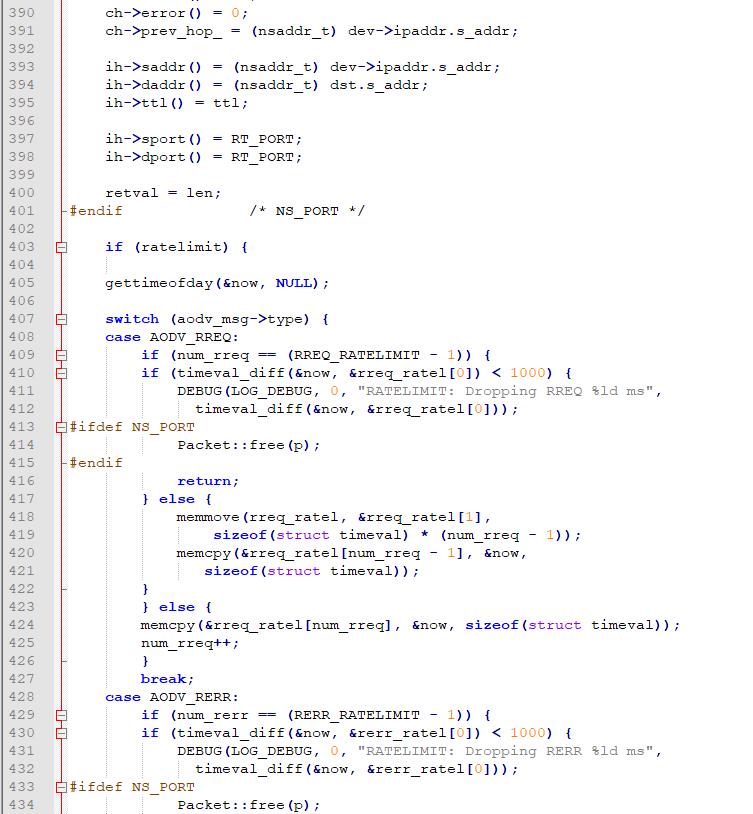
245-264 给收到的数据包先分配缓存空间，然后释放掉

266-270 检查包是否是本节点任意一个活跃接口发出的，如果是，则直接返回

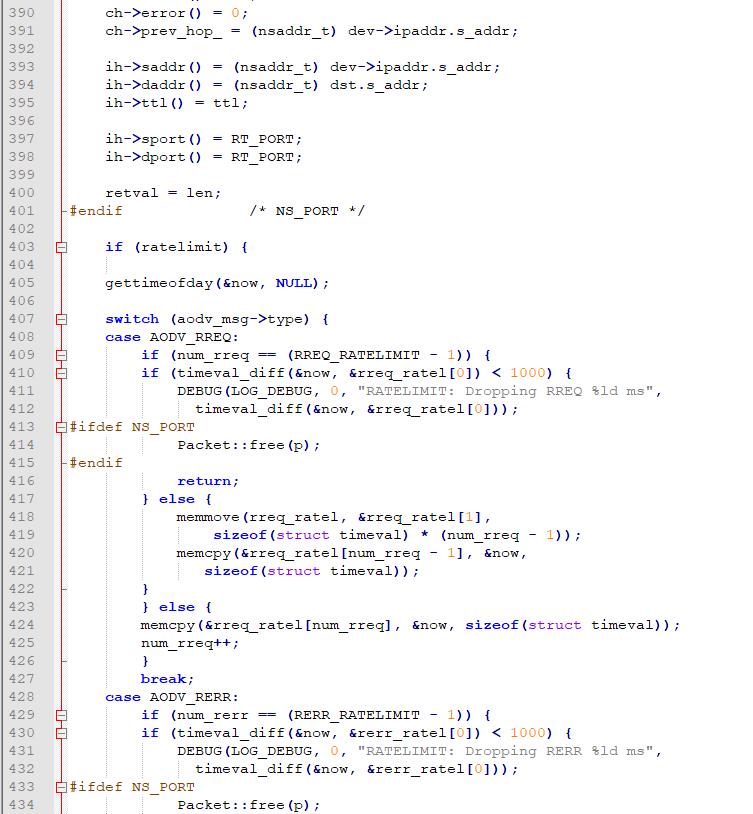
272-272 把包递交给下一函数进行处理

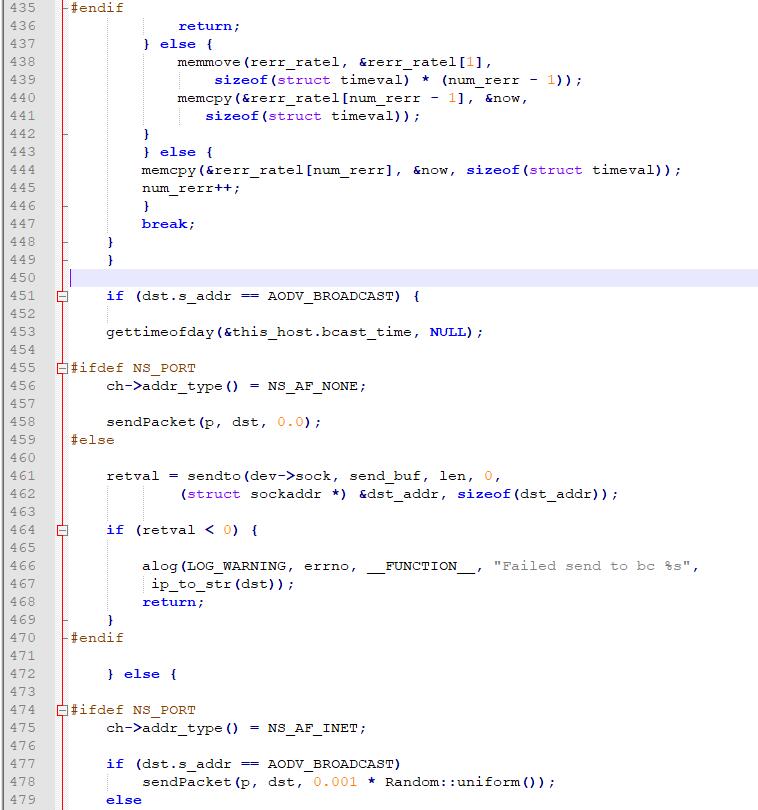
### aodv\_socket\_send函数-套接字发送函数



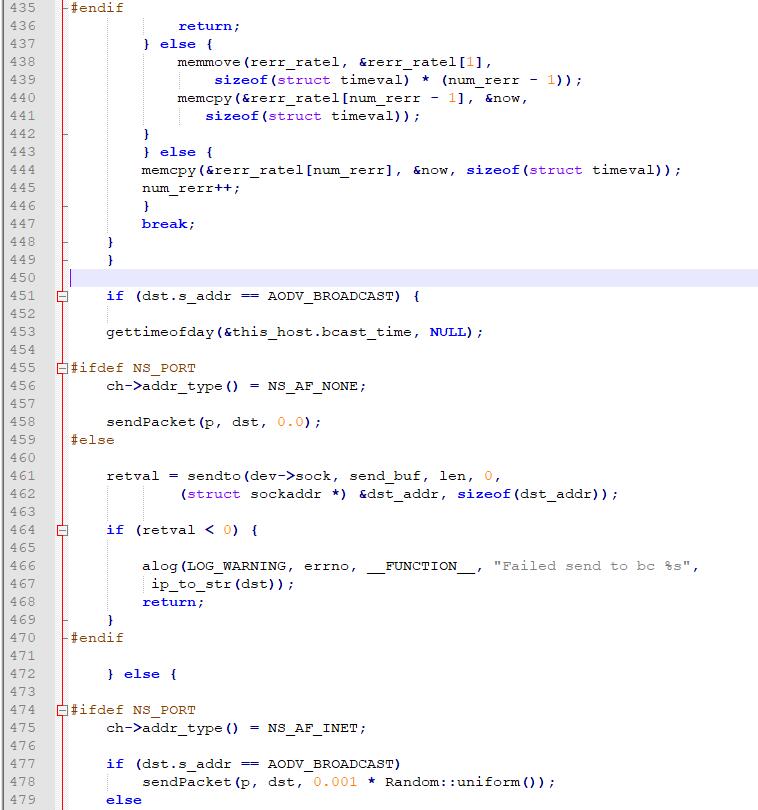


352-401 如果没有定义NS\_PORT设置头部消息和TTL，否则检查是否是RREP，不是则为包申请空间，将相关信息填入。





403-449 如果存在发送速率限制，检测包的最低传输速率是否比其类型对应的限定值小，如果小，则丢弃。如果符合则自身对应类型序列+1。





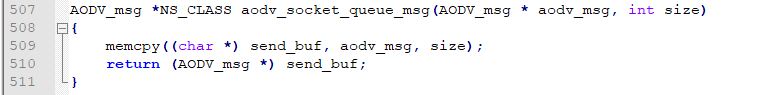
450-497 如果要发送的包是要广播的，那么更新最后广播该数据的时间，能够避免HELLO消息重复广播。

### aodv\_socket\_new\_message函数



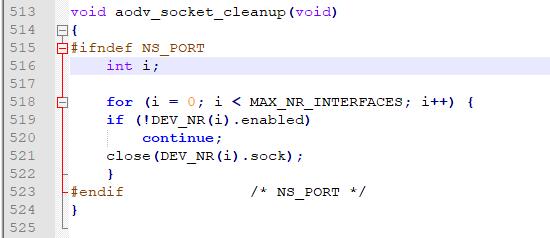
500-504 为新消息分配空间，并返回该消息的指针

### aodv\_socket\_queue\_msg函数



507-511 一个AODV消息存储在发送队列中

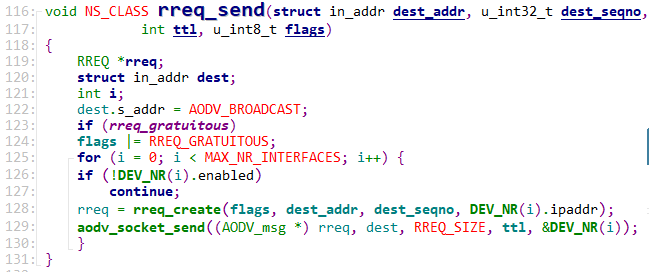
### aodv\_socket\_cleanup函数



513-525 关闭每个开放接口的套接字连接

## AODV\_rreq.c

### rreq\_send函数-发送rreq消息



125-129 向所有端口广播rreq消息

### rreq\_forward函数-转发rreq消息

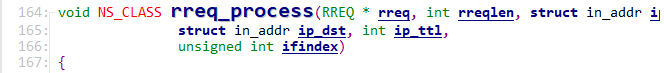


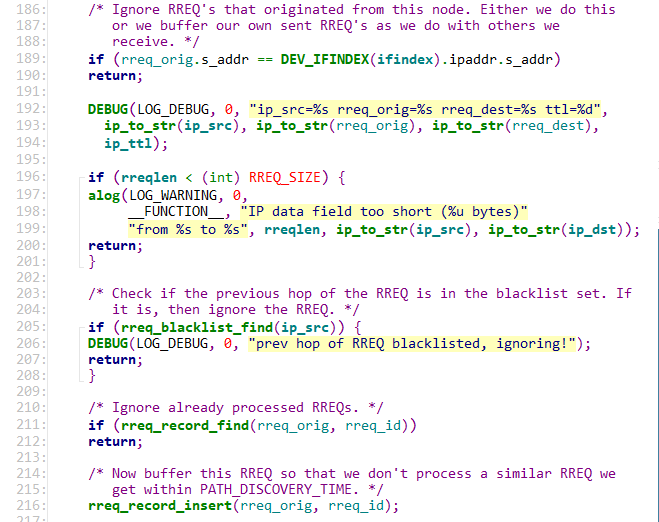
147-148 在ttl大于0之前转发接收到的消息

151-153 将接收到的rreq消息添加到接收缓存区中，并且增加跳数

157-160 在所有端口广播rreq消息

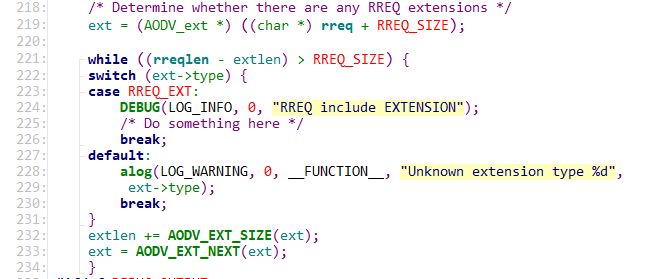
### rreq\_process函数-处理rreq消息



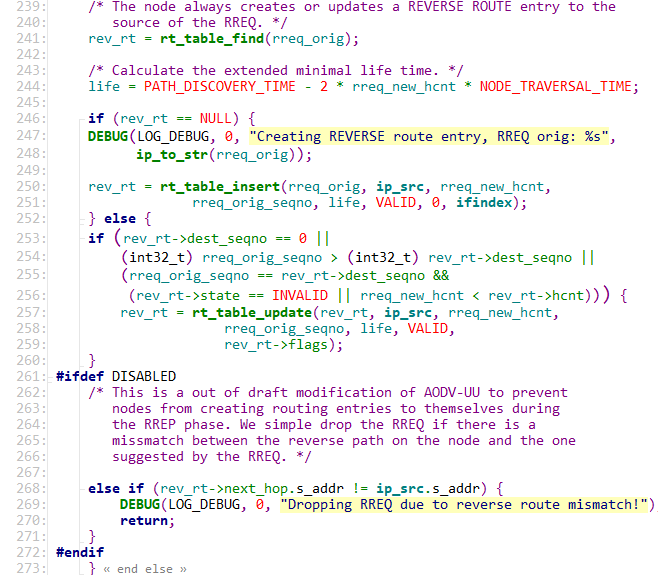


186-212 检查rreq消息，如果来自节点自身或已被加入黑名单或已有记录，则忽略该消息

216 将该消息加入record中，防止在路由发现时间内处理相同的消息



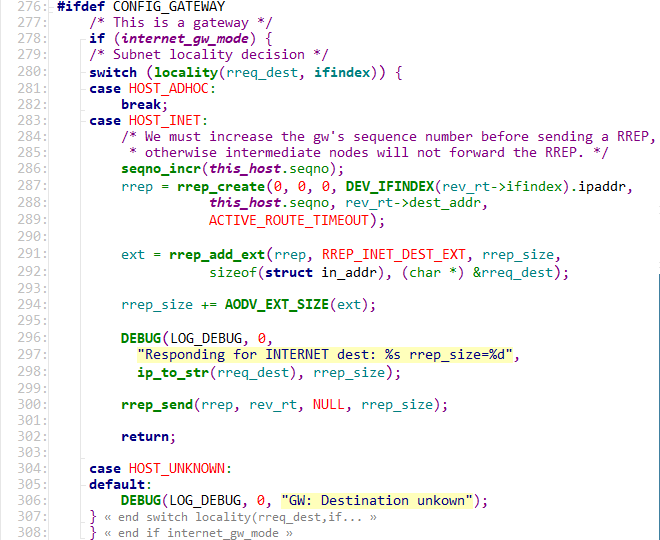
218-237 检查是否还存在RREQ拓展消息



241-273 更新反向路由，具体方法为：

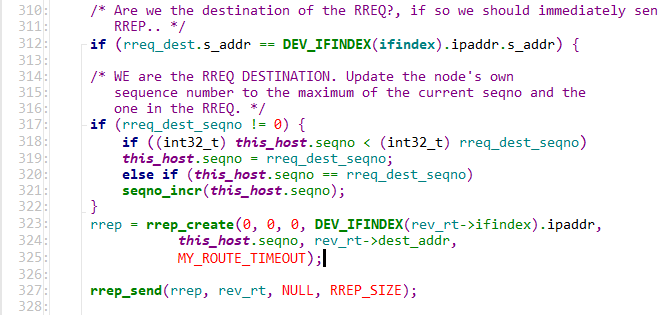
1. 如果路由表没有该rreq的源地址，则插入一条关于该源地址的路由表项；
2. 如果路由表有关于源地址的路由表项，满足下面三个条件之一则更新路由表项
3. 路由表的目的地址序列号为0；
4. Rreq的目的地址的序列号是否大于路由表项的序列号；
5. 序列号相等但是rreq跳数较少或者该路由表项因超时等失效了

268-271检查是否在RREP阶段建立了自环

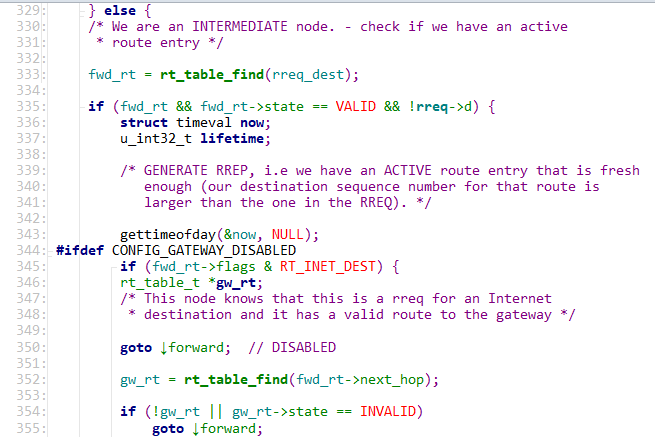


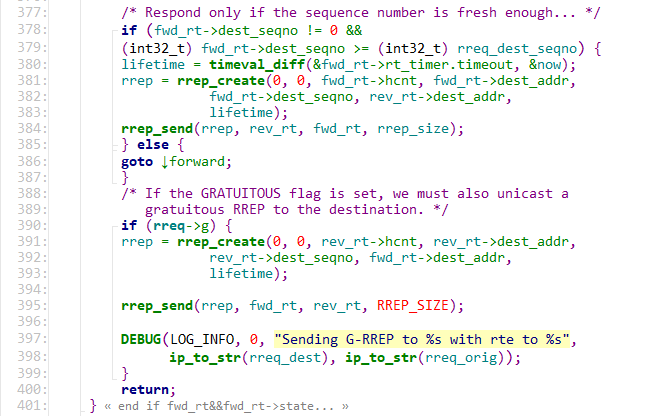
281-306 如果网关的模式，如果为HOST\_INET，则序列号自增，并创建和发送rrep消息

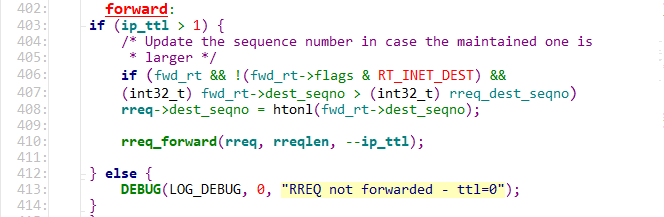
防止中间节点不转发rrep消息？



310-327 如果本节点为rreq的目标节点，更新自己的序列为rreq序列号如果自己的序列号与rreq序列目的地序列号相同，则更新自己序列号维rreq目的地序列号+1，然后回复rrep消息

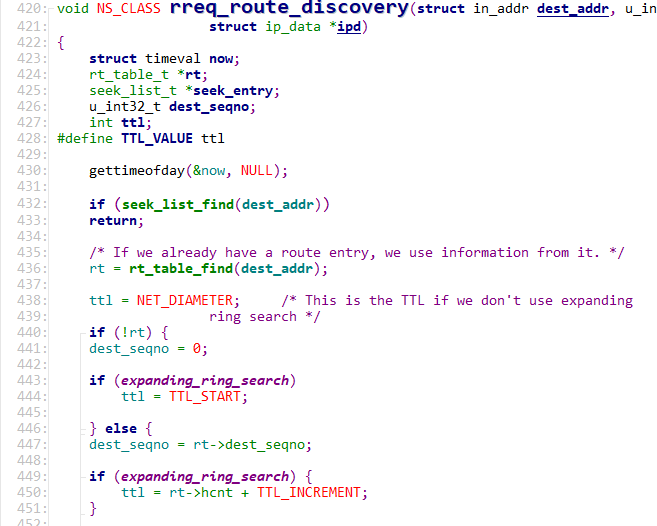


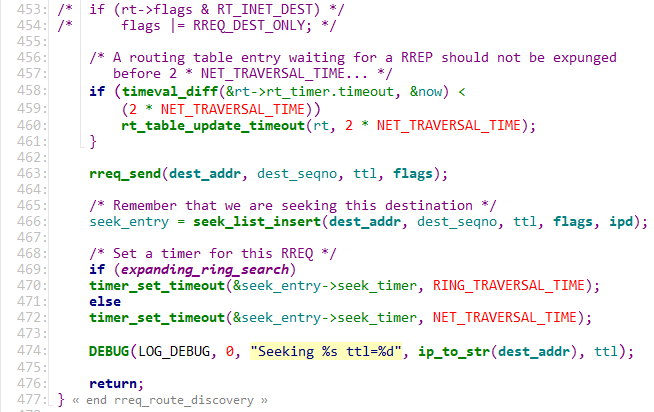




329-343,377-401，402-414如果本节点不是目标节点则查找路由表中有无该路由表项如果有有效的路由表项，当路由表的序列号比rreq目的地序列号大或者rreq的g为1的时候回复rrep否则转发ttl>1的rreq消息

### rreq\_route\_discovery函数-路由发现过程





432 如果源节点路由表中存在到目的节点的路由，则不调用AODV协议

434-463 如果没有，源节点广播rreq消息并记录在seek\_list里面

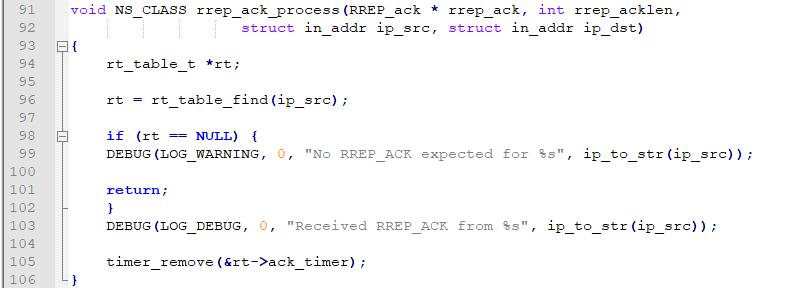
### 其他函数

|  |  |  |
| --- | --- | --- |
| 函数名 | 行数 | 说明 |
| RREQ \*NS\_CLASS rreq\_create(u\_int8\_t flags, struct in\_addr dest\_addr, u\_int32\_t dest\_seqno, struct in\_addr orig\_addr) | 61-96 | 创建rreq消息，设置jrgd四个参数值。当一个节点要产生RREQ洪泛之前，必须要增大序列号， |
| AODV\_ext \*rreq\_add\_ext(RREQ \* rreq, int type, unsigned int offset, int len, char \*data) | 98-114 | 增加rreq拓展消息 |
| void NS\_CLASS rreq\_local\_repair(rt\_table\_t \* rt, struct in\_addrsrc\_addr,struct ip\_data \*ipd) | 480-540 | 作用与路由发现函数类似 |
| NS\_STATIC struct rreq\_record \*NS\_CLASS rreq  \_record\_insert(struct in\_addr orig\_addr,u\_int  32\_trreq\_id) | 542-572 | 记录rreq消息 |
| NS\_STATIC struct rreq\_record \*NS\_CLASS rreq\_record\_find(struct in\_addr orig\_addr,u\_int  32\_t rreq\_id) | 574-587 | 查找rreq消息是否在记录中 |
| void NS\_CLASS rreq\_record\_timeout(void \*arg) | 589-595 | 销毁记录 |
| struct blacklist \*NS\_CLASS rreq\_blacklist\_insert  (struct in\_addr dest\_addr) | 597-621 | 将某目的地址加入黑名单 |
| struct blacklist \*NS\_CLASS rreq\_blacklist\_find  (struct in\_addr dest\_addr) | 623-634 | 查找目的地址是否在黑名单中 |
| void NS\_CLASS rreq\_blacklist\_timeout(void \*ar  g) | 636-643 | 销毁黑名单 |

## AODV\_rrep.c

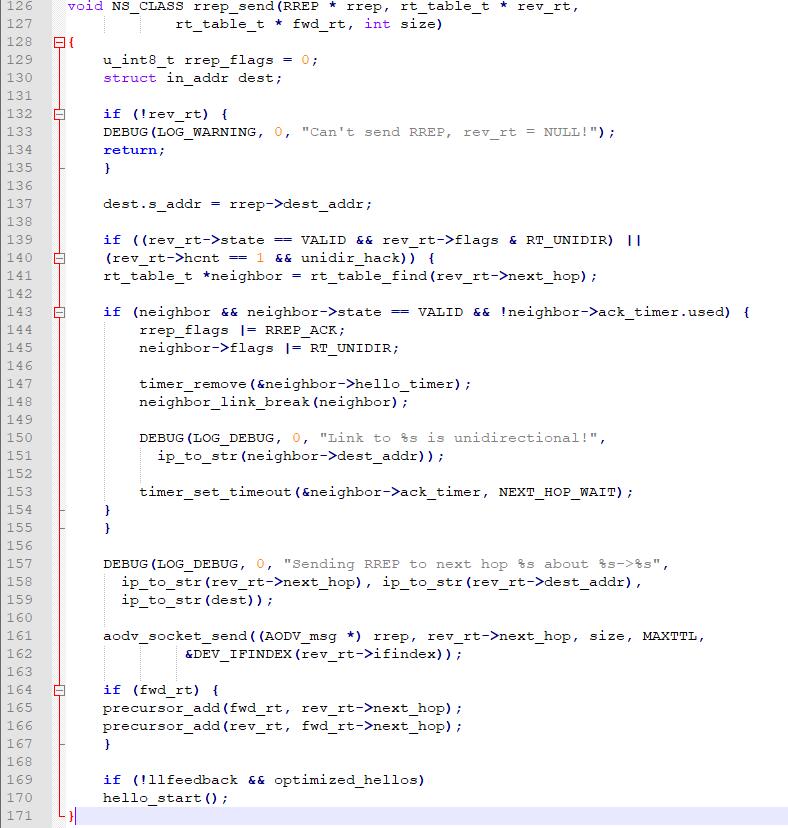
路由节点进行路由发现的时候，RREP路由控制帧会从目的节点或中间节点单播回原节点，这时候中间节点和源节点都会收到RREP帧，所以需要有RREP帧的处理函数。同时中间节点需要发送RREP帧，所以还要有生成和发送RREP帧的函数。

### rrep\_ack\_process函数



91-106 判断路由表中是否有该包的源IP相关路由项，如果没有就说明该消息非法，如果有则在控制台显示相关信息，移除ack计时器。

### rrep\_send函数-发送rrep消息



132-135 如果当前节点路由表为空，那么不能发送RREP消息

139-141 检查是否应该发送RREP\_ACK消息

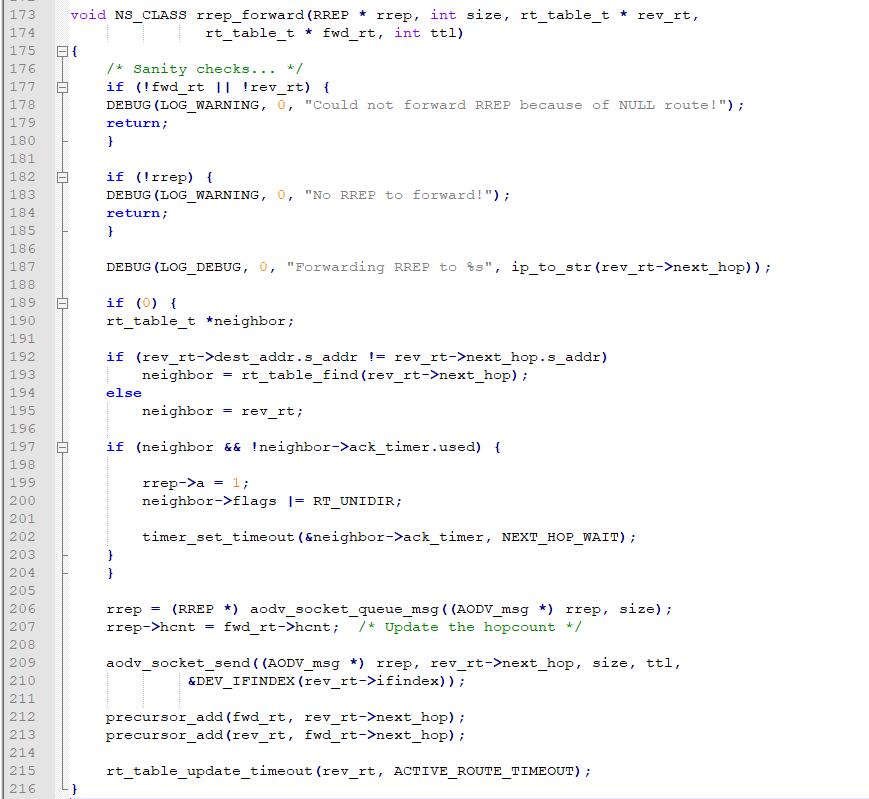
143-155 如果邻居存在并且邻居的状态是有效的，计时器还没使用，那么就将包的flag设置为RREP\_ACK，邻居标记为非单向，移除邻居的hello计时器，设置ack计时器。

157-159  更新路由表项

161-161 交由aodv\_socket\_send函数去发送消息

164-167 更新fwd\_rt和rev\_rt的前驱节点

### rrep\_forward函数-转发rrep消息



176-180 当fwd\_rt或rev\_rt不存在时，无法进行转发，因此返回；

182-185当rrep是空时，返回

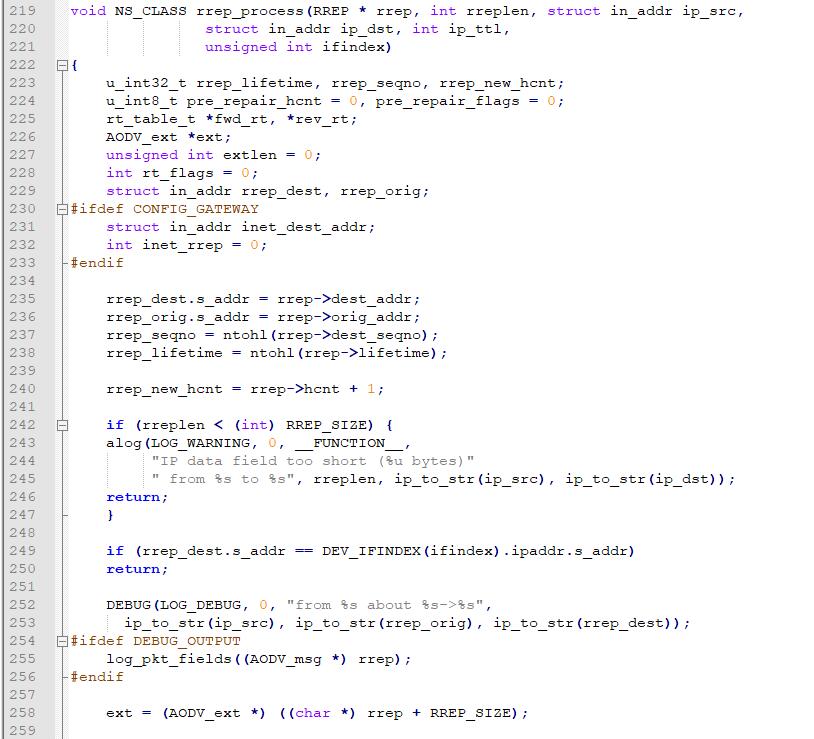
192-195 如果RREP消息的源IP地址不是相邻节点，那么我们必须找到通向源节点的下一跳IP地址。

197-204 如果我们收到邻居的RREQ控制帧，那么我们可能面临单项连接，需要将该链路设置为非单向，然后设置定时器，请求RREP\_ACK。

206-216 ，将RREP加入消息队列，交由下一函数发送，更新前驱列表，更新路由表项的有效时间

### rrep\_process函数-处理rrep消息

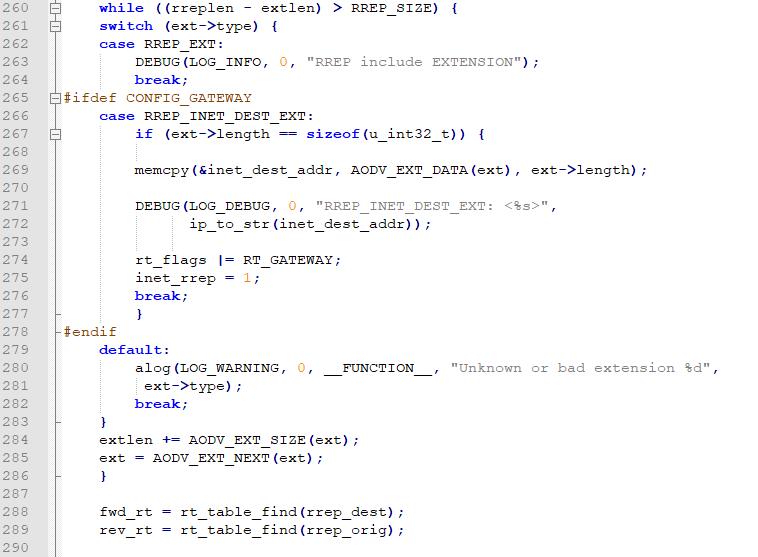
当套接字接收到RREP回复帧的时候，经过初步的过滤操作，最终会递交给这个函数进行处理。一个节点接受到RREP回复帧的时候需要判断是否是自己发出的RREQ回复的，如果不是则查询路由表项，将RREP重新单播回去。



223-240 声明变量，提取RREP回复帧中的各种消息，方便后续使用

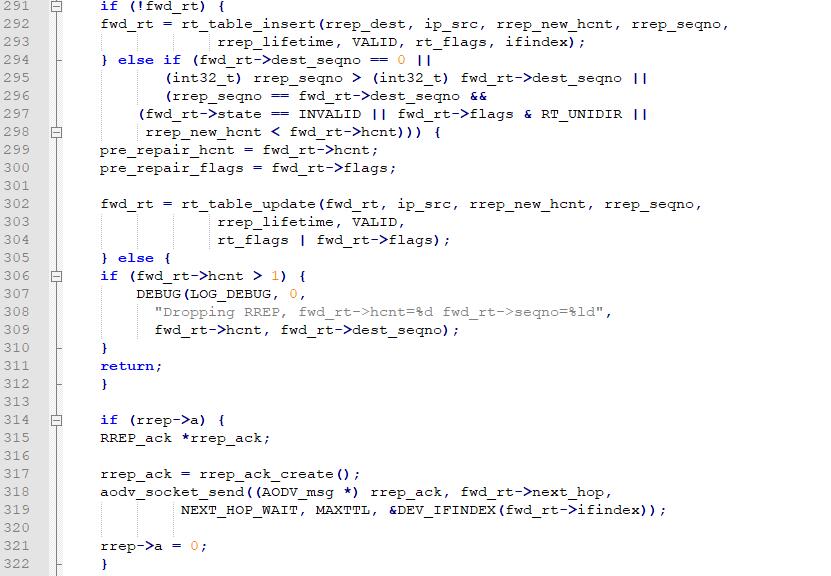
242-247 如果RREP包的长度小于限定值，显示错误信息，返回

249-250 如果是自己发出的包，直接忽略



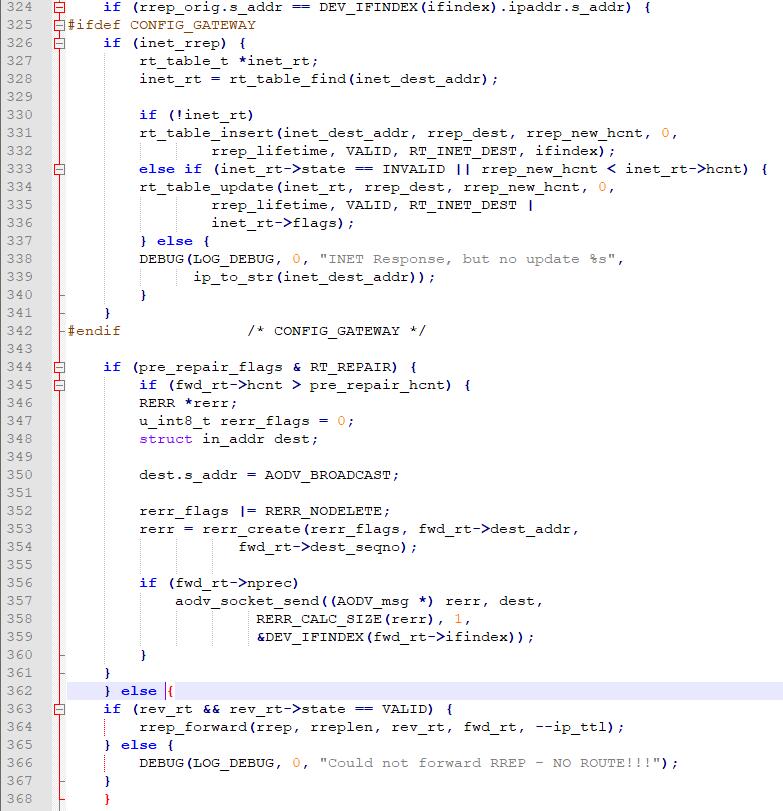
260-286 如果包的长度减去扩展长度仍大于RREP包的设定大小则进入该循环。该循环对RREP扩展和目的节点的拓展进行处理

288-290 获取有关RREP目的节点和源节点的路由表项以待后续使用



291-312 如果没有关于目的节点的表项，那么就在路由表新加入该表项，如果有而且序列号大于路由表项记录的序列号，那么更新路由表，否则忽略

314-322 检测是否需要发送RREP\_ACK，需要则发送



324-368 如果源地址就是本节点，那么如果定义了默认网关，就进行对应的更新和插入操作。如果断开的节点处于被修复状态，一个NODELETERERR应该被发送到源节点，源节点重新发起路由发现过程。如果源地址不是本节点，判断是否需要转发RREP包，需要转发则转发。

369-370 根据条件，决定是否开启HELLO帧。

### 其他函数

一些实现上与RREQ帧重复的和不是那么重要的函数以表格的形式展示，就不进行深入解读了

|  |  |
| --- | --- |
| 名称 | 功能说明 |
| RREP \*NS\_CLASS rrep\_create(u\_int8\_t flags,  u\_int8\_t prefix, u\_int8\_t hcnt, struct in\_addr dest\_addr, u\_int32\_t dest\_seqno, struct in\_addr orig\_addr, u\_int32\_t life) | 该函数通过传入的参数设置RREP包的prefix，hop count，目的IP和对应序列号，源IP和对应序列号等信息，返回一个新创建好的RREP包 |
| RREP\_ack \*NS\_CLASS rrep\_ack\_create() | 创建RREP\_ACK控制帧 |
| AODV\_ext \*NS\_CLASS rrep\_add\_ext(RREP \* rrep, int type, unsigned int offset, int len, char \*data) | 为RREP包添加扩展，并返回拓展后的包 |
| int rrep\_add\_hello\_ext(RREP \* rrep, int offset, u\_int32\_t interval) | 在RREP中包括Hello Interval扩展并返回新的偏移量 |

## aodv\_rerr.c

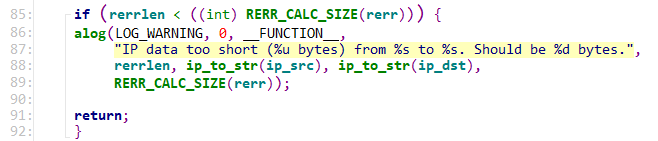
### rerr\_create函数-创建

该函数有三个形参，包括：flag，dest\_addr，dest\_sepno。C:\Users\755261024\AppData\Roaming\Tencent\Users\755261024\QQ\WinTemp\RichOle\7I29ID5EG]G5OH`J7576IJE.png

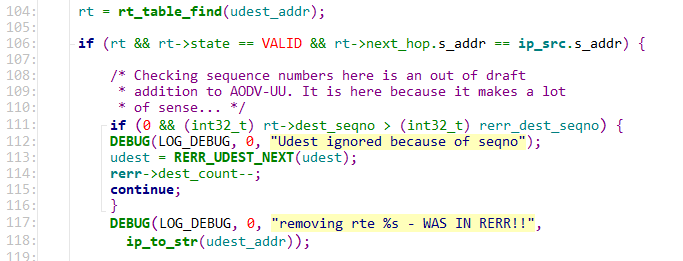
flag有0和1两个值，置1时，表示路由已经执行了错误链路的修复，前驱节点不用再删除路由；dest\_addr和dest\_seqno分别表示不可达地址的IP和序列号。

### rerr\_process函数-处理

节点收到一条rerr消息后，调用该函数。程序的流程如图所示：



85-92首先判断数据包的有效性，若收到的rerr消息，小于RERR\_CALC\_SIZE（rerr）时，说明该rerr在传输的过程中出现错误，丢弃数据包；



104-116 判断数据包有效后，节点在本地路由表中查找不可达节点。找到不可达节点后，若该节点所在路由表项依然有效。在111行，比较不可达节点的目的序列号和路由表中最新节点的序列号，如果后者大于前者，则说明接收到的rerr消息已经过期，直接丢弃。否则，117行，删除改有效路由表项。 

124-126 调用rt\_table\_invalidate函数，将包含不可达节点的表项设置为无效。

135-170 判断不可达节点是否包含前驱节点。若new\_rerr仍为NULL，生成一条新的rerr消息，将不可达节点IP加到不可达列表中。若包含一个前驱节点，将该前驱节点地址设置为rerr消息单播地址。若包含多个前驱节点，，将所有节点添加到本节点新生成的rerr消息中。

176-177 删除所有不可达节点的前驱节点



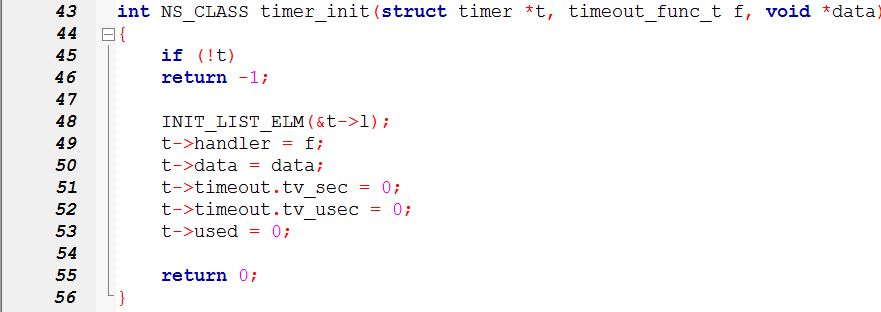
190-209 发送新生成的rerr消息。若只有一个目标节点，则单播该消息。否则，将rerr消息逐个发送到受影响的节点中。因为本段代码将ttl=1，所以仅发送给本节点的邻居节点。

### 其他函数

|  |  |
| --- | --- |
| void rerr\_add\_udest(RERR \* rerr, struct in\_addr udest, u\_int32\_t udest\_seqno) | 向rerr消息中添加一条不可达节点 |
| RERR\_UDEST\_FIRST(rerr) | 检查不可达的目的节点 |
| RERR\_UDEST\_NEXT(udest) | 查到一个不可达节点后，继续检查下一个不可达节点 |

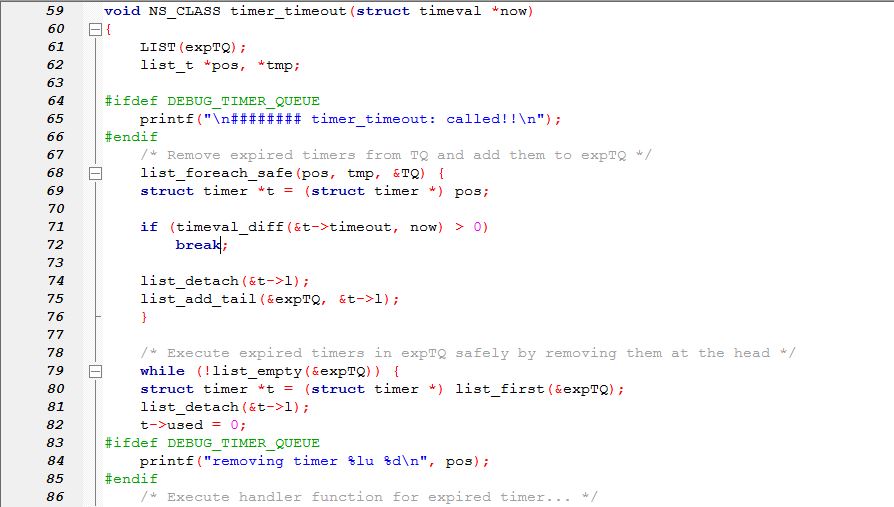
## Time\_queue.c

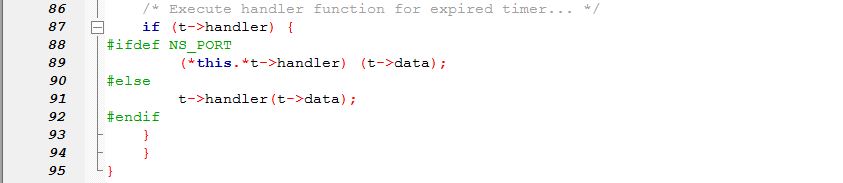
### NS\_CLASS timer\_init函数-记时器初始化函数



设定一个新的计时器，如果没有计时器的结构体便退出，如果有，便对计时器进行赋值。

### NS\_CLASS timer\_timeout函数-计时器超时，调用此函数进行清理工作





当计时器超时，调用此函数进行清理工作

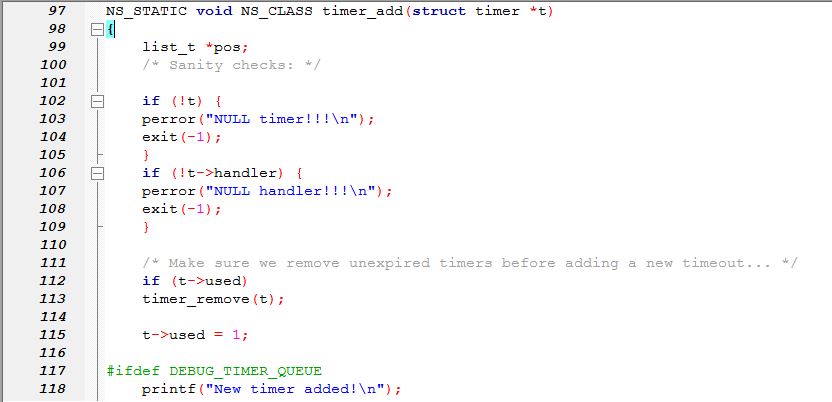
61：创建list\_t的结构体，结构体名为expTQ，并且将该结构体的两个指针赋值

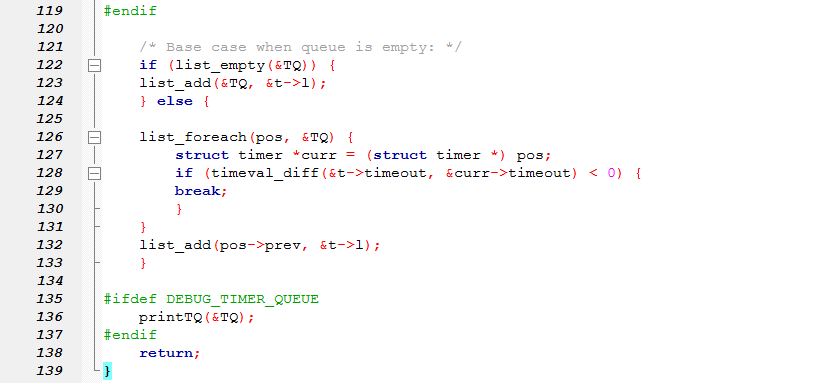
62：创建list\_t的结构体的指针，pos，tmp

68-76：遍历头指针为TQ的队列，如果此时计时器超时则退出循环，并且将整个队列的排列顺序倒置

79-94：如果队列expTQ不为空，就将expTQ的首指针赋值给结构体\*t，并将该指针从队列中删除，然后t中的uesd值设为0，

### NS\_CLASS timer\_add函数-在计时器队列中添加计时器函数





99：定义了list\_t的指针\*pos

102-105：如果t构造失败，返回-1

106-109：如果t->handler构造失败，返回-1

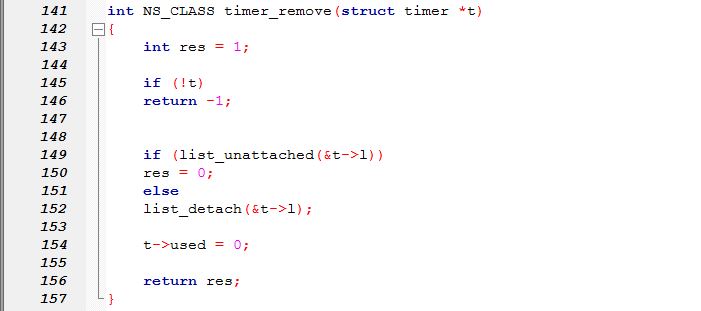
112-113：如果t->used值为1，表明此定时器已经被使用，便移除此定时器

115：used置为1，表明此定时器已经开始被使用

122-124：如果定时器为空，则直接添加一个定时器

126-131：如果定时器不为空，则遍历整个定时器队列，如果这个定时器已经超时，则退出这个循环，否则便将定时器插在定时器队列的末尾。

### NS\_CLASS timer\_remove函数-计时器队列中移除计时器



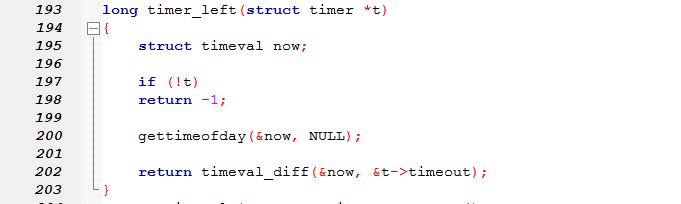
143：定义并初始化res

145-146：如果计时器t的定义错误，name便返回-1

149-150：如果这个定时器并没有添加到队列中，那么便将res值置为0

151-152：否则，便将定时器t删除，并将used值置为0，并返回res

### timer\_left函数-计算计时器剩余时间



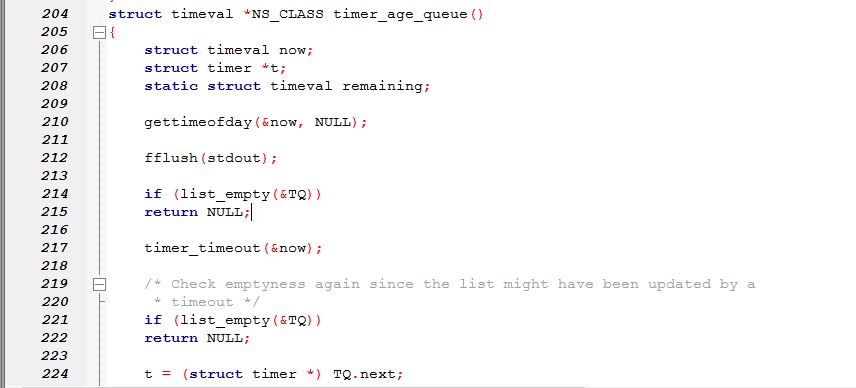
195：定义timeval结构体 now

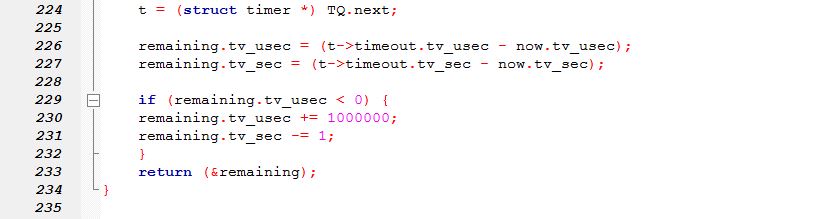
197-198：如果t的结构体构造不正确，返回-1退出

200：得到当前时间

202：返回计时器的剩余时间

### timer\_age\_queue函数-计算计时器队列的生存时间





206-208：定义两个timeval结构体，分别为now，remaining，和一个timer的结构体 t

210：获取当前时间

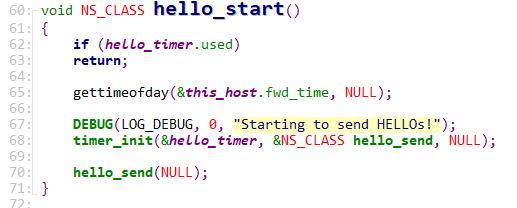
212：对标准输出流的清理，及时地打印数据到屏幕上

214-215：如果队列为空，则返回空

221-233：再次检查空白，因为列表可能已经超时更新，并对更新的值重新赋值

## aodv\_hello.c

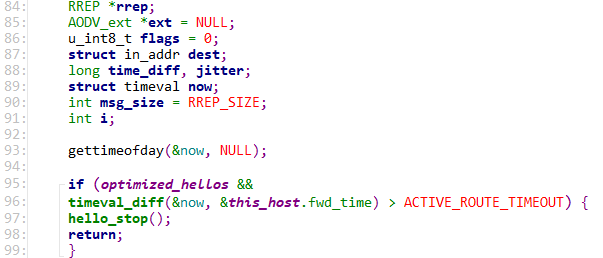
### hello\_start()函数-启动hello消息



62-63 若定时器处于开启状态，直接返回

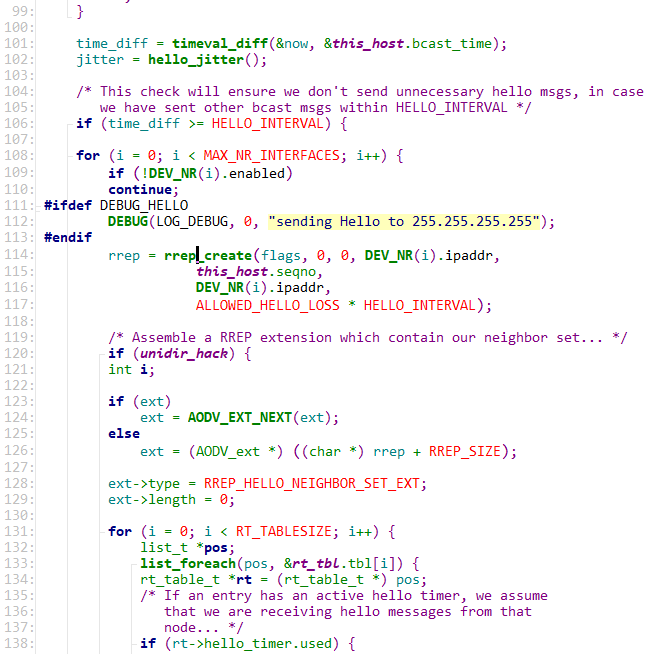
65-70 开启hello消息计时器，首先获取系统当前时间，设定DEBUG参数，表示开启发送hello消息的操作。68行，启动hello消息定时器，70行，将hello消息发送出去。

### hello\_send()函数-发送hello消息



84-89 hello消息属于rrep消息类型。所以发送函数想要发送一个hello消息，首先要构造一条rrep消息。这些参数的功能分别是：dest，属于in\_addr结构体，用于存放目的地址; now，属于timeval结构体，用于存放系统当前时间；msg\_size，整型数据，用于存放刚刚生成的rrep数据包的大小。time\_diff 和 jitter是用于计时的参数。

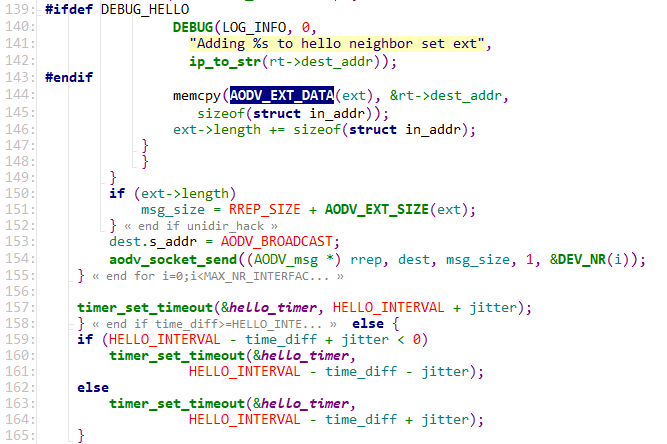
91-97：获取系统当前时间，存放于now中。计算当前时间与上次发送hello数据包的时间差，若大于路由超时时间，则说明节点和邻居节点的链接处于断开状态，调用hello\_stop函数，关闭hello消息，函数返回。



99-104：计算当前系统时间与节点上一次广播hello消息的时间，记为time\_diff。若time\_diff大于HELLO\_INTERVAL,说明未发送不必要的hello消息，否则，说明发送了不必要的消息。

104-110 查看每一个设备是否处于就绪状态，如果设备就绪且DEBUG\_HELLO已经定义，则广播hello消息并输出消息提示。

111-136 宏DEBUG\_HELLO未定义，通过调用rrep\_creat函数组装一个rrep数据包，将所有的邻居节点加到这个数据包中。



137-147：在此判断DEBUG\_HELLO, 如果已经定义，则将rt节点的所有dest\_addr添加到rrep数据包的邻居组中，否则，将dest\_addr中长度为in\_addr的结构体长度的字节，通过调用memcpy函数，添加到AODV\_EXT\_DATA中，同时更新ext的长度。

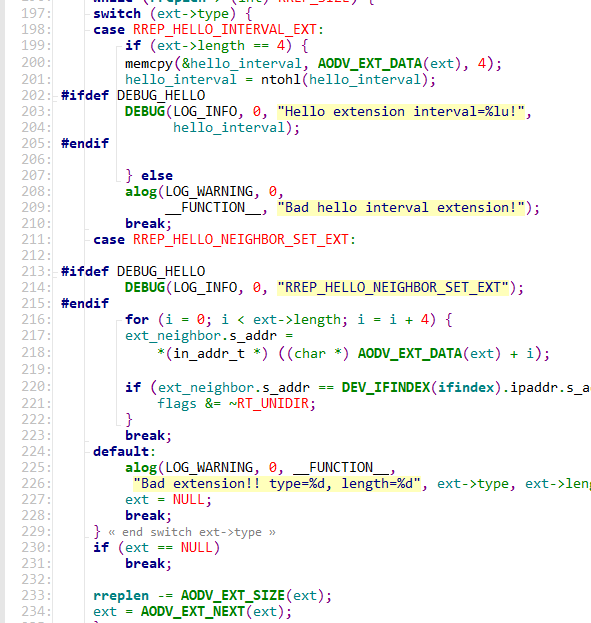
148-157：初始化数据包的大小，将组装好的rrep消息，广播给所有的邻居节点。调用time\_set\_timeout函数，设定计时器时间为HELLO\_INTERVAL + jitter；

157-162：如果两个hello消息的时间差小于设定的标准时间差，差值小于抖动值jitter，设定hello消息计时器的值为HELLO\_INTERVAL - time\_diff – jitter，否则，设为HELLO\_INTERVAL - time\_diff + jitter；

### hello\_process()函数-处理hello消息



172-191 接收hello消息，并在自己的路由表象中查找hello消息中包含的dest\_addr条目。找到对应条目后，将flag的值设为1。



197-234：如果接收到的hello消息长度有效，根据ext->type构成switch语句，从接收到的两次相邻的hello消息，计算时间间隔。同时记录rrep消息的长度。

240-246：要成功接收到来自邻居节点的三条连续hello消息才有效。收到三条hello消息，则对应的receive\_n\_hellos置1，则state置为VALID，否则，state为INVALID。

248： 计算hello消息的超时时长timeout。

250-264：如果本节点的路由表项内，没有找到和hello消息中dest\_addr相关的路由表项，则在路由表中调用rt\_table\_insert()函数添加一个新的路由表项，通过包含hello消息的rrep数据包，初始化新的路由表项。

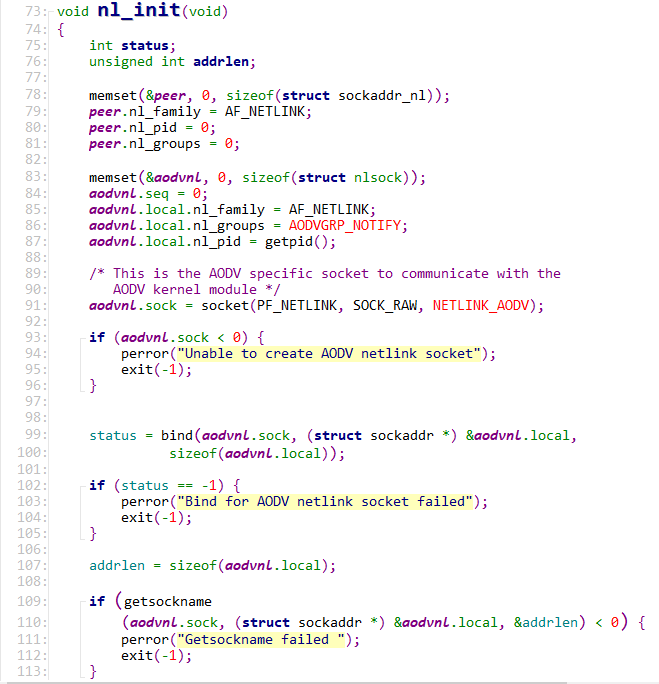
265-282 在路由表中找到对应的表项后，则通过执行hello\_update程序，增加该表项的生命周期。同时，若这条路由表项包含hello消息中的目的序列号，则表项中hello\_cnt+1，否则，通过调用rt\_table\_update()函数，将hello消息中的目的序列号添加到路由表项中。

### 其他函数

|  |  |
| --- | --- |
| long hello\_jitter() | 设置抖动值 |
| void hello\_stop() | 关闭hello消息，将hello消息的计时器从计时器队列中移除 |
| hello\_update\_timeout(rt\_table\_t \* rt, struct timeval \*now, long time) | 更新超时时间，加上处理数据包的时间 |

## Nl.c

### Nl\_init函数-初始化函数





以上为nl\_init函数第一部分主要是建立AODV套接字，此套接字负责与AODV内核模块通信。

75-87：将aodvnl，peer分别进行初始化

91：创建一个原始套接口，这是AODV特定的用于与AODV内核模块通信的套接口，socket函数第一参数为PF\_NETLINK，表明套接字遵循协议族NETLINK。第二参数SOCK\_RAW，表明采用原始套接字。第三参数NETLINK\_AODV，表明采用的协议为AODV特有的netlink协议。具体的AODV的netlink部分在/lnx/kaodv-netlink中，这部分将在3.11中具体阐述

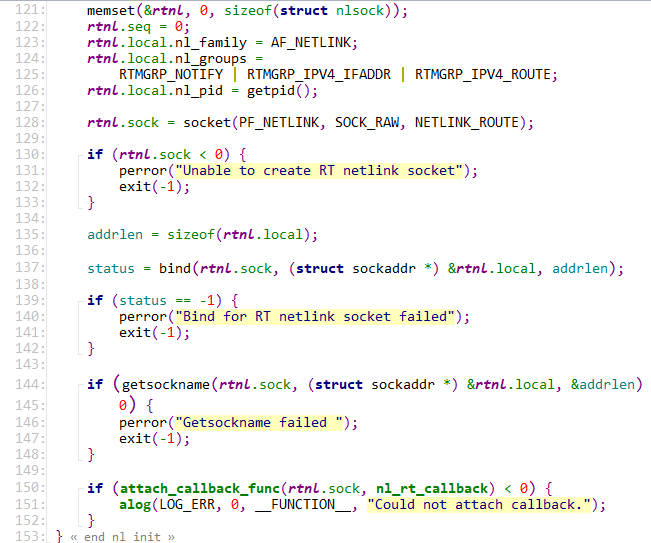
93-96：如果创建套接字失败，则输出Unable to create AODV netlink socket，并返回-1

99-100：调用bind函数，为aodvnl创建的套接字关联一个相应地址并赋给state

102-105：如果bind函数出错则输出Bind for AODV netlink socket failed，并返回-1

109-113：获取aodvnl.sock套接字的本地地址保存在aodv.local中，名字缓冲区长度保存在addrlen中，如果返回失败则返回-1退出

115-117 nl\_kaodv\_callback函数和aodv套接字放入到callback结构体里面



以上为nl\_init函数的第二部分，主要是创建的rtnl套接字

122-126：对rtnl进行初始化并赋值

128：将rtnl.sock创建为一个原始套接口，这个套接口通用路由套接口，是用于添加和的删除内核路由表项

130-133：如果这个套接口创建失败，则输出nable to create RT netlink socket并返回-1退出

137：调用bind函数，为rtnl创建的套接字关联一个相应地址并赋给state

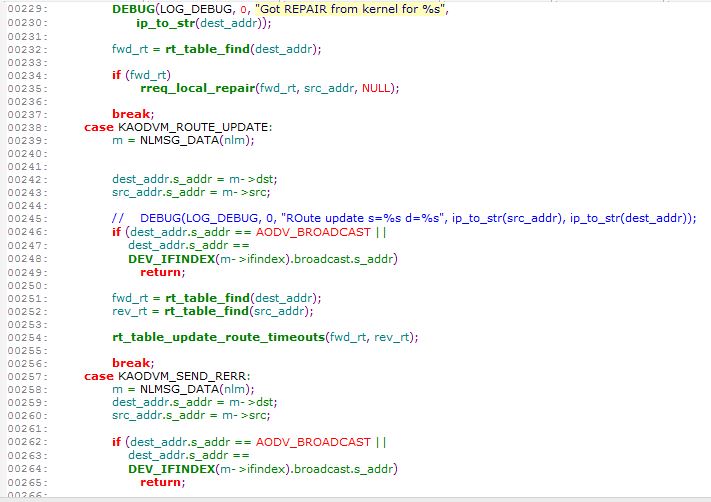
139-142：如果赋予协议地址出错，则输出Bind for RT netlink socket failed并返回-1退出

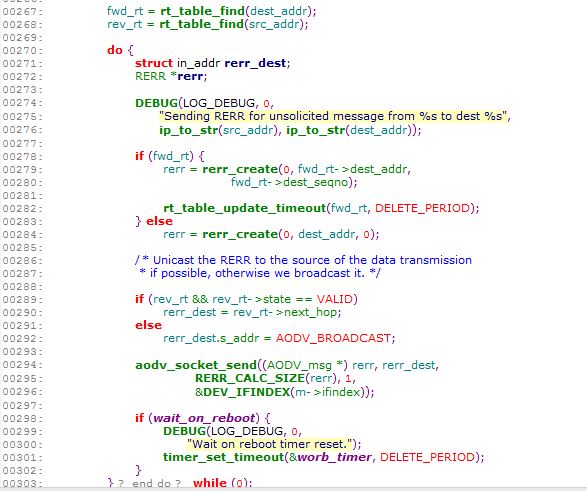
144-148：如果获取rtnl.sock的本地地址失败，则输出Getsockname failed 并返回-1退出

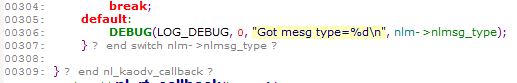
150-153：l\_kaodv\_callback函数和aodv套接字放入到callback结构体里面,如果失败，则输出Could not attach callback.

### Nl\_kaodv\_callback函数-根据aodv套接字中收到的不同命令，执行不同的回调函数











164-171：addrlen：socklen\_t类型

nlm：nlmsghdr类型的指针, nlmsghdr具体如下

/\* struct nlmsghd 是netlink消息头\*/

struct nlmsghdr {

\_\_u32 nlmsg\_len; /\* 消息长度（包括头部） \*/

\_\_u16 nlmsg\_type; /\* 消息类型 \*/

\_\_u16 nlmsg\_flags; /\* 消息标记 \*/

\_\_u32 nlmsg\_seq; /\* 序列号 \*/

\_\_u32 nlmsg\_pid; /\* 发送进程的端口号，对于内核来说是0，用户进程是socket所绑定的ID号\*/

};

其中nlmsg\_type是Linux的netlink.h下定义的四种通用消息类型

#define NLMSG\_NOOP 0x1 /\* 不执行任何动作，必须将该消息丢弃 \*/

#define NLMSG\_ERROR 0x2 /\* 消息发生错误 \*/

#define NLMSG\_DONE 0x3 /\* 标识分组消息的末尾 \*/

#define NLMSG\_OVERRUN 0x4 /\* 缓冲区溢出，表示某些消息已经丢失 \*/

#define NLMSG\_MIN\_TYPE 0x10 /\* 预留的控制信息 \*/

nlmerr：nlmsgerr类型的指针该结构体携带有错误消息

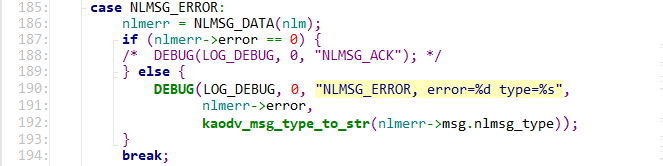
dest\_addr，src\_addr：in\_addr类型的指针，分别表示目的IP地址和源IP地址

m：kaodv\_rt\_msg\_t类型的指针，在kaodv-netlink.h中定义，携带aodv路由消息

rt, fwd\_rt, rev\_rt：rt\_table\_t类型的指针，代表路由表项

179-180：如果读取失败，便返回空

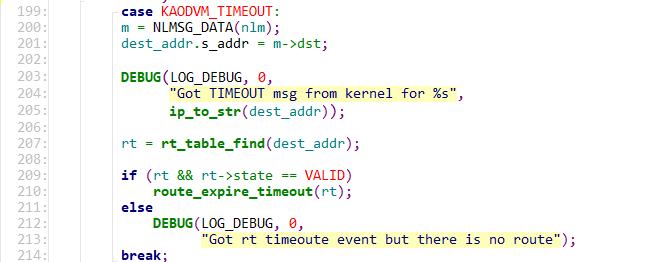
184：根据内核套接字nlm接受到的不同命令类型选择不同的处理方式



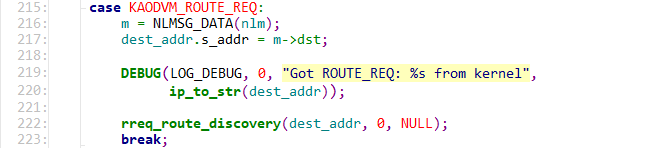
185-197：如果nlm的类型为NLMSG\_ERROR，且nlmerr->error值不为0，则输出错误信息，包括错误号和错误类型



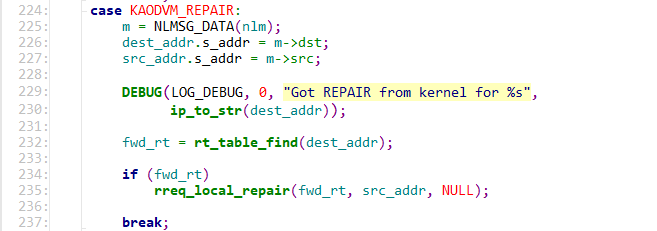
196-198如果消息类型为KAODVM\_DEBUG则输出输出附带的调试信息



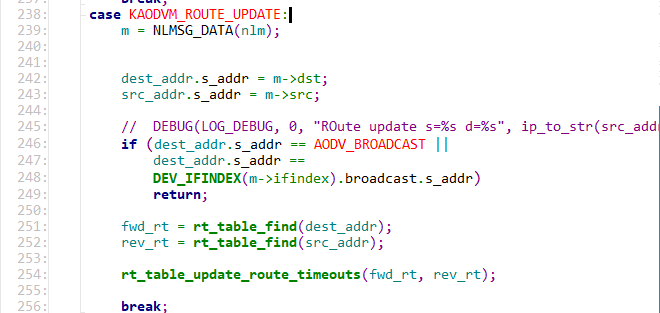
199-214如果消息类型为KAODVM\_TIMEOUT,如果路由表中目的地的路由表项有效，则将它设置为超时无效状态



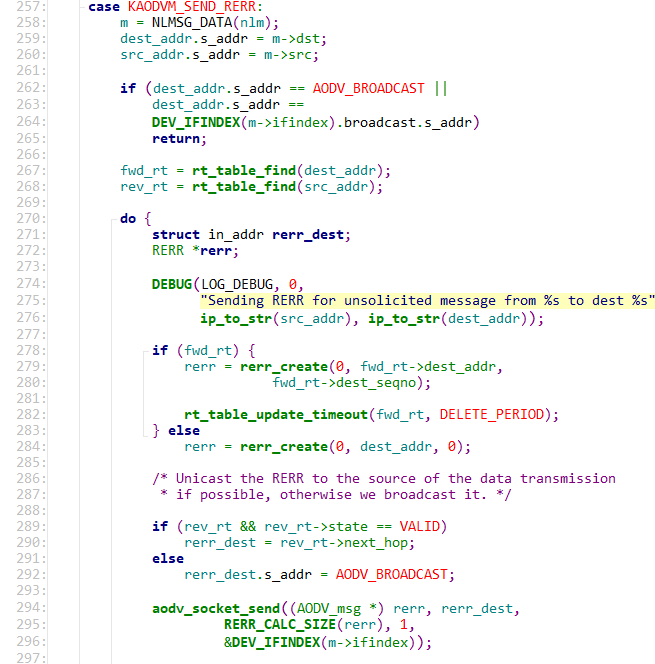
215-223如果消息类型为KAODVM \_ROUTE\_REQ类型，开始路由发现过程

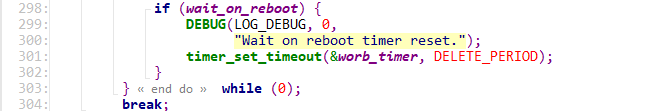


224-237 如果类型为 KAODVM \_REPAIR且目的地址在路由表中，开始本地修复过程



238-256：如果类型为 KAODVM\_ROUTE\_UPDATE，如果要广播目的地址，那么便返回，否则刷新路由表中路由表项的过期时间





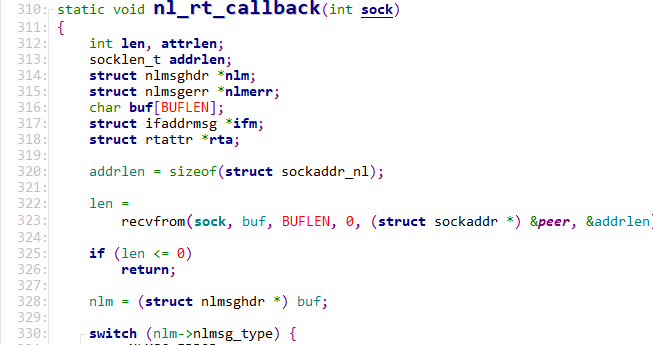
257：如果类型为KAODVM\_SEND\_RERR

262-268：如果目的地址是广播地址，那么便返回空，否则便找到目的地址和源地址

270-287：进入循环，如果路由表有关于目标地址的路由表项，则创建一个rerr 的消息类型，并且更新该路由表项的过期时间，否则便创建一个将序列号置为0的rerr类型的消息

289-303：如果源地址路由表项类型的状态是VALID，那么便将这个rev\_rt的目的地址指向下一跳，否则便广播目的地址，也就是说如果可能我们将单播RERR从源地址的数据传输，否则我们将广播它

### Nl\_rt\_callback函数-根据路由套接字中收到的不同命令，执行不同的回调函数



312-318：addrlen：socklen\_t类型，用于表示地址长度

nlm：nlmsghdr类型的指针

nlmerr：nlmsgerr类型的指针

ifm：ifaddrmsg类型的指针，该类型的消息携带添加,删除或者接收一个和接口相关的IP地址的信息

struct ifaddrmsg {

\_\_u8 ifa\_family;

\_\_u8 ifa\_prefixlen; /\* 前向长度 \*/

\_\_u8 ifa\_flags; /\* 标记为 \*/

\_\_u8 ifa\_scope; /\* 地址范围 \*/

\_\_u32 ifa\_index; /\* 连接展示 \*/

};

rta：rtattr类型的指针，创建或者删除一个特定的网络接口，或者从一个特定的网络接口上获得信息会使用到该类型

struct rtattr

{

unsigned short rta\_len; /\* 选项长度 \*/

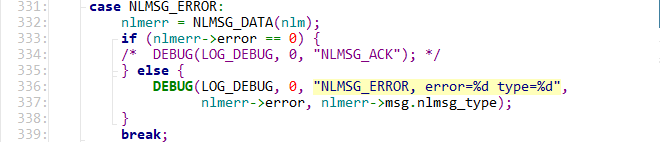
unsigned short rta\_type; /\* 选项类型 \*/

/\* Data follows \*/

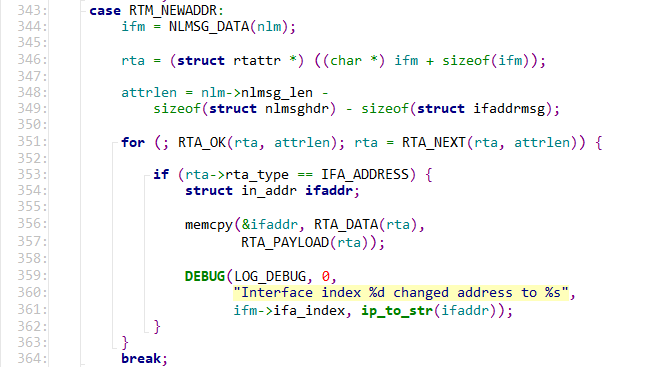
};

322-326：如果recvfrom调用失败，则返回空

330：根据nlm->nlmsg\_type即路由套接字收到的不同命令选择分支语句

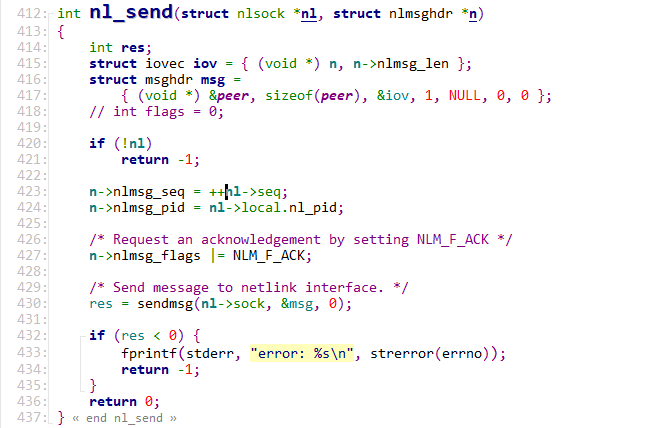


331-339：如果值为NLMSG\_ERROR且nlmerr->error的值不为0，则输出错误编号和类型



343-364如果值为RTM\_NEWADDR，则首先为ifm、rta、attrlen赋初值。在rta队列中寻找谁的rta\_type的值为IFA\_ADDRESS，则将RTA\_PAYLOAD(rta)长度的RTA\_DATA(rta)复制到&ifaddr，即将对应端口的IP地址改为消息指定的IP地址。

### Nl\_send函数-发送数据包给内核模块



415-420：如果nl指针不为空，那么便对指针n进行赋值，否则便返回-1

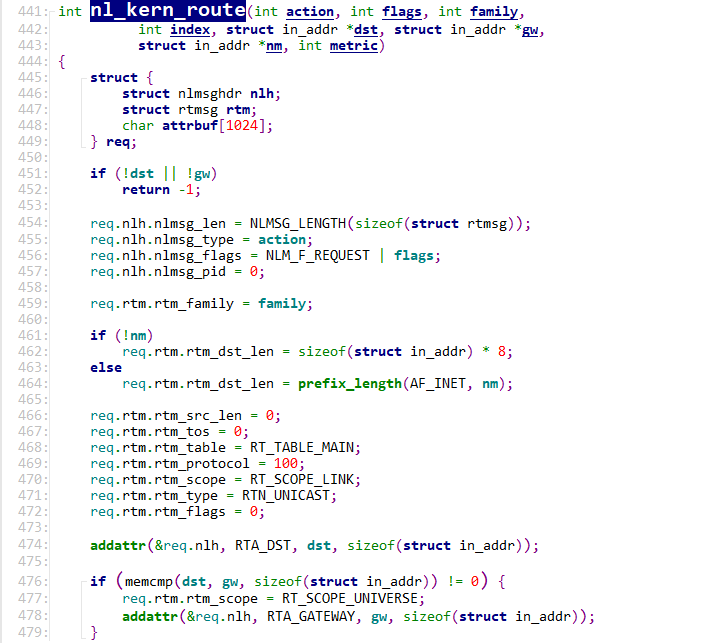
423-424序列号加1，设置目标的pid

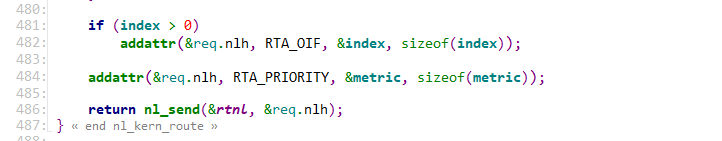
427：通过设置NLM\_F\_ACK来请求确认

430：发送消息到netlink接口，res为负，表明函数调用失败

432-435：如果函数sendmsg（）调用失败，便打印出error，退出函数，返回-1

### Nl\_kern\_route函数-操作内核路由表，包括增加、修改、删除路由表条目等操作





445-449：定义结构体req，包括一个nlmsghdr结构体,一个rtmsg结构体和一个char数组

451-452：如果没有目标IP地址或没有网关IP地址，则退出函数，返回-1

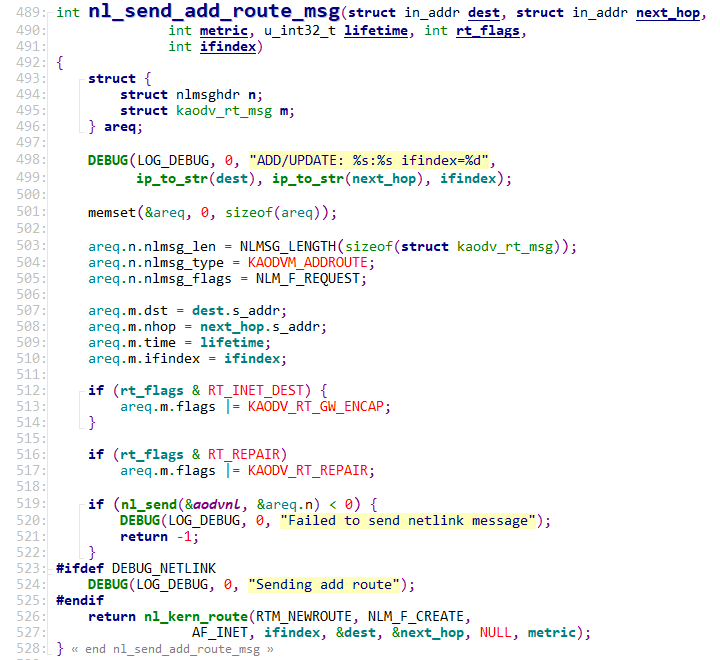
454-459：对req的成员变量进行赋值

461-462：如果nm的值为0，则req.rtm.rtm\_dst\_len便重新赋值，并再添加一个RTA\_GATEWAY类型的路由表项

481-483：如果index的值大于0便添加一个类型为RTA\_OIF的attribute

484：添加一个类型为RTA\_PRIORITY的attributer

### Nl\_send\_add\_route\_msg函数-向内核发送一个添加一个路由信息的消息，包括目的地地址、下一跳，生存期，路由状态标志，网络接口等



492-496：定义areq的结构体,包括一个nlmsghdr结构体和一个kaodv\_rt\_msg结构体（在kaodv-netlink.h中定义）

501：清空areq结构体

503-510：为areq的结构体进行赋值

512-513：如果rt\_flags右起第四位为1，则对areq.m.flags右起第一位置为1

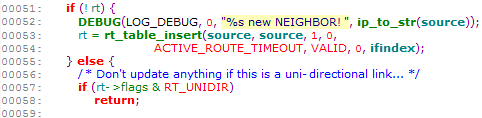
516-517：如果rt\_flags右起第二位为1，则对areq.m.flags右起第二位置为1

519-521：如果nl发送失败，则打印消息后退出函数返回-1

* 1. Aodv\_neighbor.c

**3.10.1更新活跃的邻居节点（更新来自非HELLO AODV控制消息的邻居）（void NS\_CLASS neighbor\_add）**

 代码如下：

 043-045 行代码定义了变量类型，now的作用是记录当前时间； rt作用是用作路由表（初始值设置为NULL）；seqno作用是序列号（初始值设置为0）。

051-055行代码判断路由表上是否存在sourse指向的目的节点。若不存在，利用rt\_table\_insert()函数传入参数、添加路由信息。

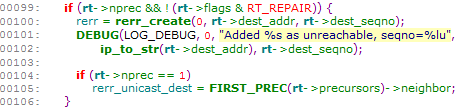
 056-059 行代码判断如果为单向链路直接返回，不用更新路由信息。

067-068 行代码表示如果路由信息是存在且活跃的，则需要用ALLOWED\_HELLO\_LOSS \* HELLO\_INTERVAL来添加对应路由的活跃周期。

### 3.10.2邻居节点链路中断情况（void NS\_CLASS neighbor\_link\_break）

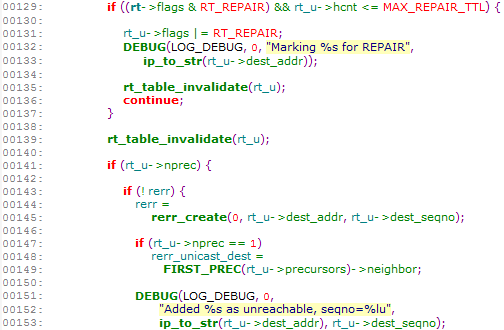
084-086 行代码判断路由表是否为空若为空则直接返回。

087-092 行代码判断若rt->hcnt跳数不为1时，则不是直接邻居节点。将目的地址和跳数传入函数并返回。

 096 行代码若判断函数为跳数为1的直接邻居节点，输出链路断开信息并将目的地址传入。同时将对应的路由表置为无效化。

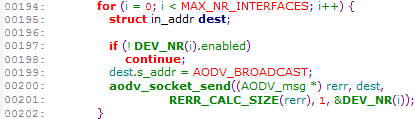
099-106行代码，除去路由被回复的情况，创建一个路由错误信息rerr。调用rerr\_create()函数来传入目的地址和序列号。

109-110 行代码用来清除了路由前驱列表。

129-152 行代码用来检测路由表的下一跳条目中知否存在不可达的目标。如果条目是不可达的，应包含在RERR中。

131-137 行代码，将损失的链接标记为要修复。

139-153 行代码将损失的链接标记为要修复的情况下，在对其他所有不可达条目应执行相同的操作。

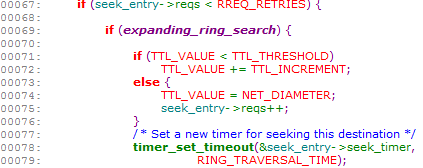
194-202 行代码表示只在不可达路由结点的先驱结点的接口上传输RERR。

## 3.11 Aodv\_timeout.c

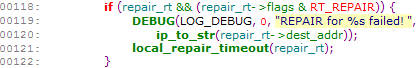
### 3.11.1定义定时器到期时调用的超期函数

059-061 行代码检查完整性。

063-065 行代码记录当前时间并写在日志中。

067-079 行代码当不满足超时的条件时，设置一个新的计时器来寻找目的地址。

088-090 行代码AODV应该使用二进制指数来退避RREP等待时间。

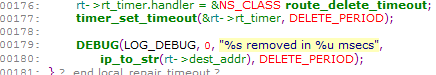


118-122 行代码保证了当路径被修复后，节点返回超时信号。

### 3.11.2.定义本地修复超时处理函数

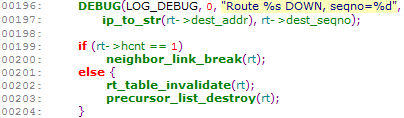


137-140 行代码默认了目的地，并清除RERAIR标志。



176-181 行代码表当修复超时，清除可能在排队中的数据包，并记录到日志中。

### 3.11.3.设置路由过期超时函数



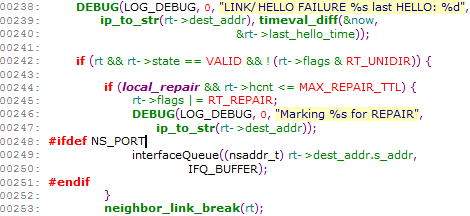
196-204 行代码当路由消息长时间没有收到对应的HELLO()函数消息时，设置这条路由超时，将其断开，并从路由表里删除。

### 3.11.4.路由删除函数设置



214-217行代码进行路由表完整性检测。从路由表中删除路由信息，并记录在日志中。

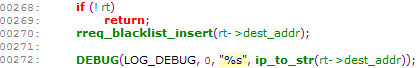
### 3.11.5.HELLO消息超时函数



238-253 行代码表示当我们停止接收来自邻居节点的HELLO()信息时，这个函数就会被调用。Hello消息超时函数与路由超时功能基本一致。当我们在修理路线时，将其标记（标志位设置为REPAIR）进行维修.

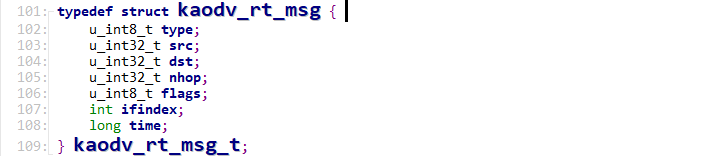
缓冲区内容来自接口队列的数据包。

### 3.11.6响应超时函数



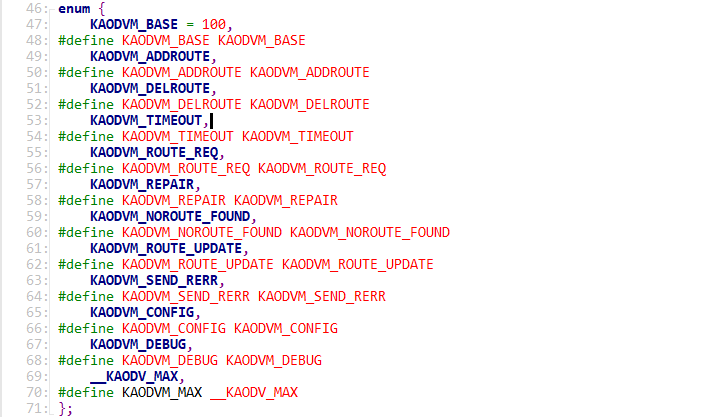
268-272 行代码，当RREP传输失败时（即缺少RREP\_ACK）时，添加到黑名单设置中。

## kaodv-netlink.h

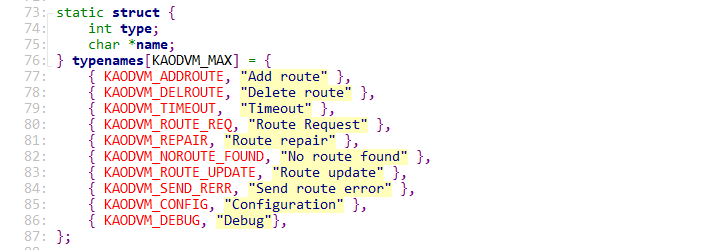
kaodv-netlink.h定义了AODV协议有关的数据结构



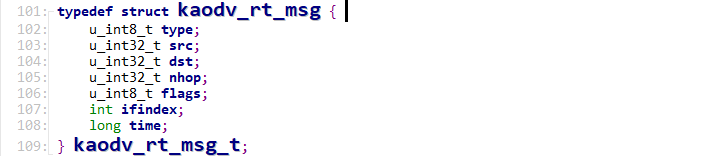
33 定义协议号为没有使用过的数字MAX\_LINKS-1，这样可以使用使用一个特有的NETLINK套接字



46-70 枚举了内核与用户空间交换的消息类型



73-87 定义了消息类型到消息名称的字典，为输出错误信息做准备

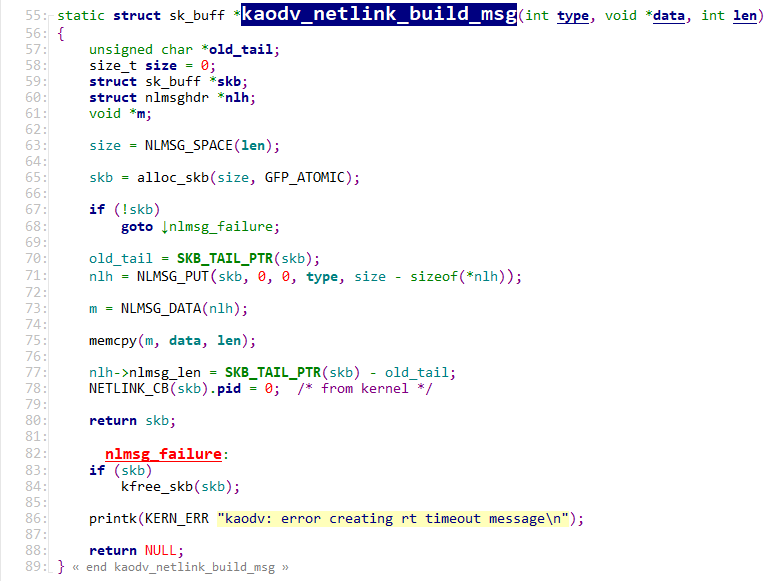


101-109定义了AODV协议路由信息的格式

## kaodv-netlink.c

本文件实现了关于AODV协议netlink机制相关的函数

### 3.13.1kaodv\_netlink\_build\_msg-创建sk\_buff结构体



59 skb是一个sk\_buff类型的指针，sk\_buff（socket buffer）结构是linux网络代码中重要的数据结构，它管理和控制接收或发送数据包的信息

Sk\_buff的结构如下

struct sk\_buff {

struct sk\_buff \*next;

struct sk\_buff \*prev;

struct sock \*sk;

struct skb\_timeval tstamp; /\* Time we arrived，记录接收或发送报文的时间戳\*/

struct net\_device \*dev; /\*通过该设备接收或发送，记录网络接口的信息和完成操作\*/

struct net\_device \*input\_dev; /\*接收数据的网络设备 \*/

struct net\_device \*curlayer\_input\_dev;

struct net\_device \*l2tp\_input\_dev;

union {

struct tcphdr \*th;

struct udphdr \*uh;

struct icmphdr\*icmph;

struct igmphdr \*igmph;

struct iphdr \*ipiph;

struct ipv6hdr\*ipv6h;

unsigned char \*raw;

} h; //传输层报头

union {

struct iphdr \*iph;

struct ipv6hdr\*ipv6h;

struct arphdr \*arph;

unsigned char \*raw;

} nh; //网络层报头

union {

unsigned char \*raw;

} mac; //链路层报头

unsigned int len, //len缓冲区中数据部分的长度。

data\_len, //data\_len只计算分片中数据的长度

mac\_len, //mac头的长度

csum; //校验和

\_\_u32 priority;

\_\_u8 local\_df:1,

cloned:1, //表示该结构是另一个sk\_buff克隆的

ip\_summed:2,

nohdr:1,

nfctinfo:3;

\_\_u8 pkt\_type:3,

fclone:2,

ipvs\_property:1;

\_\_be16 protocol;

\_\_u32 flag; /\*packet flags\*/

unsigned int truesize; //这是缓冲区的总长度，包括sk\_buff结构和数据部分

atomic\_t users;

unsigned char \*head, //指向缓冲区的头部

\*data,// 指向实际数据的头部

\*tail, //指向实际数据的尾部

\*end;//指向缓冲区的尾部

};

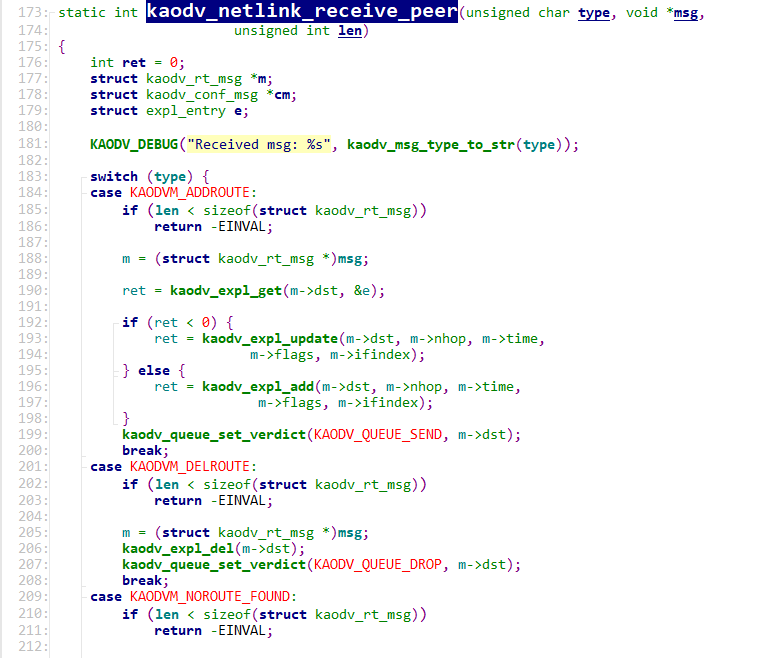
60 nlh是一个nlmsghdr类型的指针

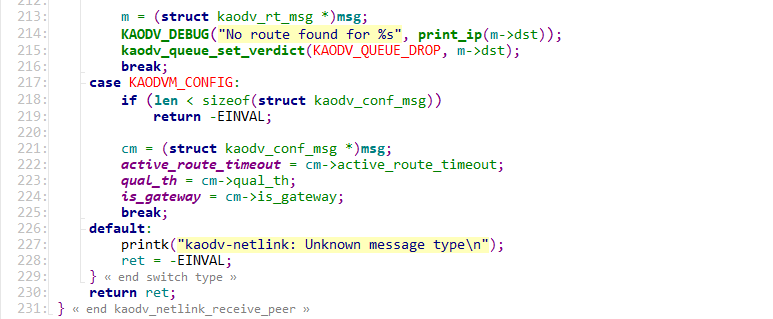
65 alloc\_skb函数分配缓冲区，包括两个缓冲区，一个是数据缓冲区，一个是缓冲区的描述结构sk\_buff

67-68,82-88 如果alloc\_skb失败，则释放skb内存并返回null

70-80 设置skb的各项值，需要注意的是，由于是从内核发出，pid应设置为0

### 3.13.2 kaodv\_netlink\_receive\_peer-内核收到消息后的处理房产税的方式





177 m是一个kaodv\_rt\_msg的指针，携带有路由信息

178 cm是一个kaodv\_conf\_msg的指针，携带有AODV协议设置信息

179 e是一个expl\_entry的指针，这是一个由AODV协议维护的路由信息

184-200 如果消息类型是KAODV\_ADDROUTE，查找AODV协议的路由表是否有关于目的地的路由表项，如果有，则更新路由表项，否则增加路由表项。并交给内核队列裁决模块

201-208如果消息类型是KAODV\_DELROUTE类型，则删除AODV协议维护的路由表中的路由表项，并交给内核队列裁决模块

209-216如果消息类型是KAODV\_NOROUTR\_FOUND类型，输出没有发现前往目的地的路由并交给内核队列裁决模块

217-225如果消息类型是KAODV\_CONFIG类型则将kaodv\_conf\_msg指针指向的参数修改

|  |  |  |
| --- | --- | --- |
| 函数名 | 行数 | 说明 |
| kaodv\_netlink\_send\_debug\_msg | 91-103 | 输出调试信息 |
| kaodv\_netlink\_send\_rt\_msg | 105-120 | 向应用空间广播路由信息 |
| kaodv\_netlink\_send\_rt\_update\_msg | 126-148 | 向应用空间广播路由更新信息 |
| kaodv\_netlink\_send\_rerr\_msg | 150-171 | 向应用空间广播端口路由信息 |
|  |  |  |

# 4 AODV仿真

## 4.1仿真思路：

仿真平台是NS2.35，ubuntu16.04。仿真分为构件的扩展阶段，仿真阶段，仿真结果分析阶段。

1. 编写脚本文件：因为NS2库中包含aodv协议，所以可以直接开始编写.tcl脚本文件。在脚本文件中，需要配置仿真网络的拓扑结构，确定链路的基本特性，节点间通信所需要的路由协议，通信节点的个数等。同时，还需要绑定端设备的协议，设置仿真场景的参数和传输负载，仿真总时长等。最后，设置nam对象，trace对象，其中nam对象是演示网络运行动画的工具，trace对象用于记录仿真过程发生的所有事件。
2. 执行脚本文件：脚本名称为aodv.tcl，使用ns aodv.tcl命令执行脚本程序，脚本成功执行后，会在同一文件夹目录下生成\*.nam和\*.tr文件，记录仿真结果。
3. 分析trace文件：由于trace文件比较大，所以在分析过程中，编写gawk程序进行仿真数据的处理，主要是计算分组投递率、路由发起频率、路由开销、平均时延四个显示路由性能的数据。绘图使用的是ubuntu自带的gnuplot绘图工具。

## 4.2 获取trace文件

### 4.2.1生成仿真环境

在终端~/ns-allinone-2.35/ns-2.35/indep-utils/cmu-scen-gen/路径下输入命令：



该命令创建一个具有50个移动节点，10对通信连接，每秒钟发送2个分组的以cbr为业务源的通信场景文件cbr-50n-10c-2p。

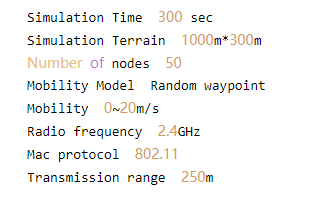
在终端~/ns-allinone-2.35/ ns-2.35/indep-utils/cmu-scen-gen/setdest/路径下输入命令：



该命令创建一个具有50个节点、节点在每个地点停留0秒(即不停留)、最大移动速度20m/s,仿真时间300秒，长1000米，宽300米的移动场景文件scene-50n-0p-20M-300t-1000-300。

### 4.2.2 编写aodv.tcl脚本

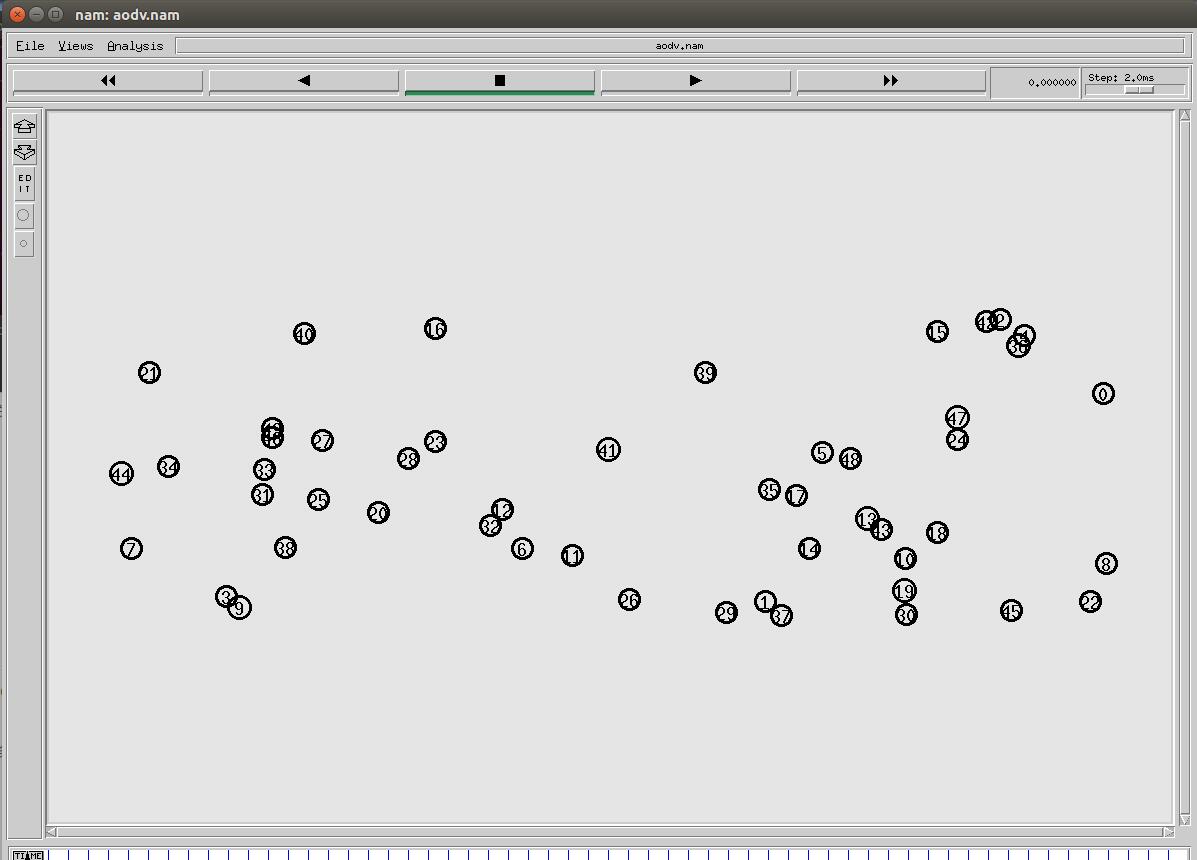
想要调用NS2中的AODV协议，需要编写脚本设置仿真的场景。脚本中的仿真参数如下图所示：



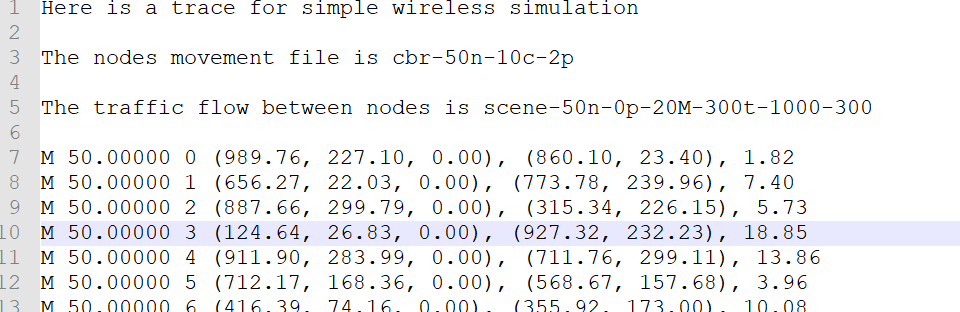
### 4.2.3 执行aodv.tcl脚本

将以上生成的通信场景文件cbr-50n-10c-2p和移动场景文件scene-50n-0p-20M-300t-1000-300，以及aodv.tcl放到同一目录下（属于NS2.35的子目录），在该目录下执行ns aodv.tcl命令，运行脚本。脚本运行完成后，会在同一目录下生成aodv.tr和aodv.nam两个文件。其中仿真的数据结果保存在aodv.tr文件中。仿真结果和trace文件中数据格式如下图所示：

仿真效果图：



数据格式：



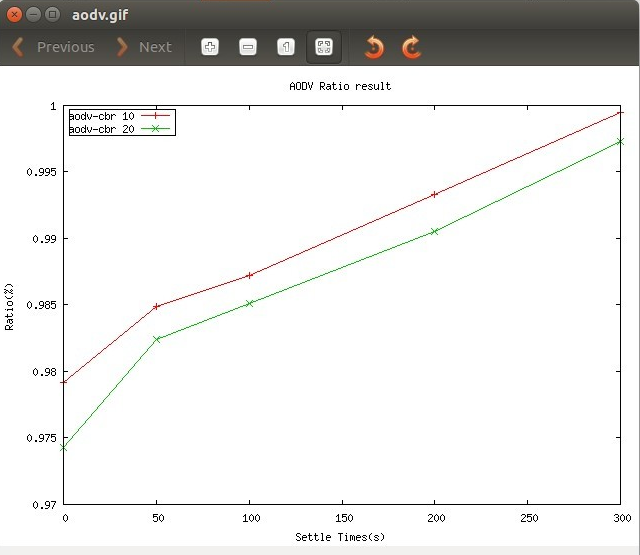
**4.3 分析trace文件**

生成的trace文件一般都比较大，所以使用average.awk, frequency.awk，getRatio.awk, delay.awk, load.awk, run六个脚本程序对数据进行处理，获取分组投递率、路由发起频率、路由开销、平均时延四个表征协议性能的参量。

仿真过程中，选择每个CBR源产生UDP数据包，每个源每秒钟发送2个数据包，每个数据包的大小为512字节。实验中，对CBR源有10个和20个两种情况进行仿真对比。得到的结果如下：

**4.3.1 分组投递率** =

分组投递率（Packet delivery ratio）是源节点传输层接收分组的数目（即应用层产生分组的数目）与目的节点传输层发送节点对的数目（即应用层产生分组数目）的比值，这一指标反映了网络的吞吐量，表明路由协议的有效性和适应网络变化的性能。对于路由协议，分组投递率越高越好。通过trace文件，分组投递率（Ratio）和节点停留时间（Settle Time）的关系如下图所示。



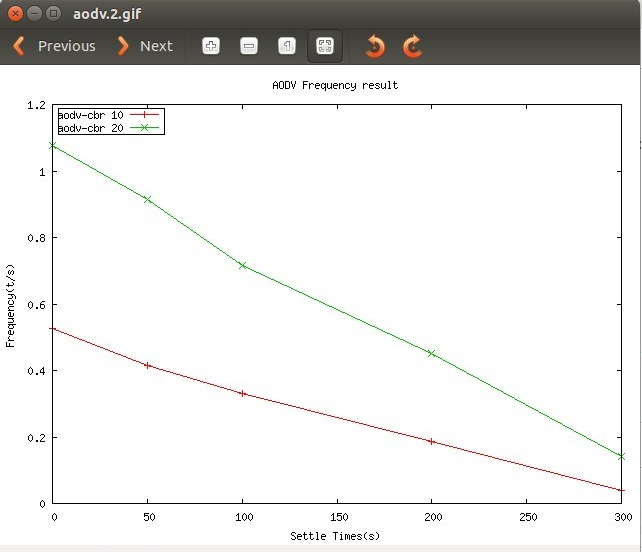
由图可知：

1. 随着节点停留时间逐渐增大，两种情况的分组投递率都增大，节点间路由出现断裂的情况减少。当停留时间为50s左右时，分组投递率的增长趋势发生明显变化，说明在这一过程中，节点与节点之间的路由信息变化较为剧烈；超过50s后，增长趋势趋于平稳，说明节点与节点之间的路由信息基本稳定，保持分组投递率的增长一直到仿真实验300秒全部结束。
2. CBR源较少的网络要比要比CBR源较大的网络分组投递率高。说明，减小CBR源的个数，可以提高数据包的分组投递率。

4.3.2 路由发起频率 =

路由发起频率（Route Discovery Frequency）是源节点发起路由发现的次数和实验的仿真时间的比值，这一指标反映了路由的有效性，同时也影响着路由的开销。对于路由协议来说，路由发起频率越低越好。

路由发起频率（Frequency）与节点停留时间（Settle Time）的关系如下图所示



由图可知：

1. 随着节点停留时间的增大，节点的路由发起频率逐渐减少。说明当节点的移动速度较慢时，节点与节点之间的路由信息基本保持稳定，路由出现断裂的情况较少，因此路由的发起频率较小；当节点的移动速度较快时，节点与节点之间的路由信息容易发生断裂，导致路由发起频率也随之增加。
2. CBR源较多的网络中，路由发起频率明显大于CBR源少的网络。说明CBR源多的网络中，路由易发生断裂。

4.3.3 路由开销 =

路由开销全称为归一化路由开销（Normalized Routing Load）是用于路由发现和路由维护的分组总数和接受到的数据报个数的比值。对于路由协议，路由开销越低越好。

归一化路由开销（Normalized Load）与节点停留时间（Settle Time）的关系如下图所示：



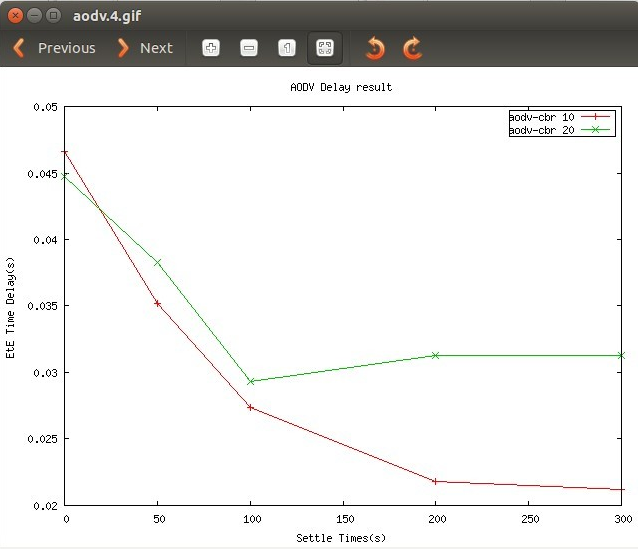
由图可知：

1. 随着节点停留时间的增大，归一化路由开销逐渐减小。说明当节点移动速度较慢时，在接收到的所有数据包中，用于路由发现和维护的分组减少，路由开销也减小。可以观察到，路由开销-节点停留时间曲线与路由发起频率-节点停留时间曲线的趋势大致是相同的，因此可以得出结论，路由发起频率和路由开销是正相关的。
2. CBR源多的网络，其路由开销也大。说明产生数据多的网络中，由于路由状态的不稳定，其用于发现和维护路由的分组较多。

4.3.4 平均时延 =

平均端到端时延（Average end-to-end Delay）是每一个数据包接收时间和发送时间差值的总和与接收到数据包个数的比值。它反映了世纪网络应用中的时间特性，它包括节点进行路由查找和在端口排队是数据分组在缓冲区的延迟，同时也包括MAC层进行重传以及分组传播的时间。对于网络协议来说，平均端到端时延越小越好。

平均端到端时延（EtE Time Delay）与节点停留时间（Settle Time）的关系如图所示：



由图可知：

1. 随着节点停留时间的增大，平均端到端的时延先减少，在停留时间为100s左右时，CBR源为20的网络，平均时延发生较大变化，平均时延改为增大；而CBR源为10的网络，平均时延的减小趋势变小，但依旧减小。这说明，对于产生数据包较多的网络，当节点的移动速度较快时，随着移动速度的减慢，平均时延减小。但当超过某一阈值后，随着移动速度的减慢，网络中的平均时延增大；对于产生数据包较少的网络，随着节点移动速度的减慢，平均端到端的时延总体成减小趋势。
2. 在节点高速移动的网络中，产生数据包较多的网络的时间特性要优于数据包少的网络。但是在节点低速移动的网络，产生数据包少的网络要有优于产生数据包多的网络。

# 结束语

经过以上的分析，我们可以发现：在Ad Hoc网络中，AODV协议操作起来比较简单，可以通过在发送数据包时使用序列号避免出现环路，网络扩展性好，适合网络变化情况较多的环境。该协议既包含路由表机制，还有按需路由机制。同时，AODV协议也包含以下缺点：没有考虑路由负载，协议开销较大，尤其是在节点快速移动的情况下，网络拓扑会发生急剧变化，协议的开销也随之剧增，从而导致端到端时延，数据包分组投递率等性能下降。