**网络协议栈分析与设计课程大作业**

|  |
| --- |
| **AODV无线自组网按需**  **距离向量路由协议代码分析** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 学号 | 姓名 | 班级 | 负责模块 | 成绩 | |  | 叶上维 | 软网1604 | 1、阅读全篇RFC文档  2、内核模块功能解读主力  3、用户模块功能解读  4、1/3源代码分析 |  | |  | 徐文博 | 软网1604 | 1、阅读全篇RFC文档  2、内核模块功能辅助解读  3、用户模块功能解读  4、1/3源代码分析  5、aodv仿真实现 |  | | 201605054 | 杨韬 | 软网1604 | 1、阅读全篇RFC文档  2、内核模块功能辅助解读  3、用户模块功能解读  4、1/3源代码分析  5、文档整合和审阅 |  | |

目录

[1. 基本原理 5](#_Toc534051422)

[1.1 概述 5](#_Toc534051423)

[1.2 特点 5](#_Toc534051424)

[1.3 AODV控制帧 5](#_Toc534051425)

[1.3.1 RREQ路由请求帧 6](#_Toc534051426)

[1.3.2 RREP路由应答帧 7](#_Toc534051427)

[1.3.3 RERR路由错误帧 7](#_Toc534051428)

[1.3.4 RREP\_ACK路由应答确认帧 8](#_Toc534051429)

[1.4 AODV用户态工作流程 8](#_Toc534051430)

[1.4.1 路由发现过程 8](#_Toc534051431)

[1.4.2 路由维护和错误控制 10](#_Toc534051432)

[1.4.3 拥塞控制 10](#_Toc534051433)

[1.4.4 序列号机制 10](#_Toc534051434)

[1.4.5 前驱节点机制 10](#_Toc534051435)

[1.5 用户空间和内核模块的交互 11](#_Toc534051436)

[2. 代码介绍 12](#_Toc534051437)

[3. 内核空间函数解读 13](#_Toc534051438)

[3.1 Linux钩子函数简介 13](#_Toc534051439)

[3.2 内核空间对数据包处理流程 14](#_Toc534051440)

[3.3 kaodv-mod.c 14](#_Toc534051441)

[3.3.1 kaodv\_hook-aodv-钩子函数 15](#_Toc534051442)

[3.3.2 kaodv\_update\_route\_timeouts-更新expl\_entry结构和通知用户进程更新用户空间的维护的路由表 18](#_Toc534051443)

[3.3.3 其他函数 18](#_Toc534051444)

[3.4 kaodv-netlink.c 19](#_Toc534051445)

[3.4.1kaodv\_netlink\_build\_msg-创建sk\_buff结构体 19](#_Toc534051446)

[3.4.2 kaodv\_netlink\_receive\_peer-内核收到消息后的处理的方式 20](#_Toc534051447)

[3.4.3其他函数 21](#_Toc534051448)

[3.5 kaodv-expl.c 21](#_Toc534051449)

[3.6 kaodv-ipen.c 21](#_Toc534051450)

[3.6.1 ip\_pkt\_encapsulate-IP数据包封装 22](#_Toc534051451)

[3.6.2 ip\_pkt\_dncapsulate-IP数据包解封装 23](#_Toc534051452)

[3.6.3 其他函数 23](#_Toc534051453)

[3.7 kaodv-queue.c 23](#_Toc534051454)

[3.7.1 kaodv\_queue\_set\_verdict-队列裁决函数 24](#_Toc534051455)

[4. 用户态函数解读 25](#_Toc534051456)

[4.1 用户态全局变量 25](#_Toc534051457)

[4.1.1 路由表项 25](#_Toc534051458)

[4.1.2 全局链表结构 25](#_Toc534051459)

[4.1.3 全局目的地链表结构 26](#_Toc534051460)

[4.1.4 全局定时器队列 26](#_Toc534051461)

[4.2 main函数 26](#_Toc534051462)

[4.3 main函数文件内其他函数以及变量的定义： 30](#_Toc534051463)

[4.4 AODV\_socket.c 31](#_Toc534051464)

[4.4.1 aodv\_socket\_init函数-套接字初始化函数 31](#_Toc534051465)

[4.4.2 aodv\_socket\_read函数-套接字读取 33](#_Toc534051466)

[4.4.3 aodv\_socket\_process\_packet函数-处理数据包函数 35](#_Toc534051467)

[4.4.4 recvAODVUUPacket函数-接收AODVUUPacket 36](#_Toc534051468)

[4.4.5 aodv\_socket\_send函数-套接字发送函数 37](#_Toc534051469)

[4.4.6 其他函数 39](#_Toc534051470)

[4.5 AODV\_rreq.c 40](#_Toc534051471)

[4.5.1 rreq\_process函数-处理rreq消息 40](#_Toc534051472)

[4.5.2 rreq\_route\_discovery函数-路由发现过程 43](#_Toc534051473)

[4.5.3 其他函数 45](#_Toc534051474)

[4.6 AODV\_rrep.c 45](#_Toc534051475)

[4.6.1 rrep\_ack\_process函数 45](#_Toc534051476)

[4.6.2 rrep\_send函数-发送rrep消息 46](#_Toc534051477)

[4.6.3 rrep\_forward函数-转发rrep消息 47](#_Toc534051478)

[4.6.4 rrep\_process函数-处理rrep消息 47](#_Toc534051479)

[4.6.5 其他函数 50](#_Toc534051480)

[4.7 aodv\_rerr.c 51](#_Toc534051481)

[4.7.1 rerr\_process函数-处理 51](#_Toc534051482)

[4.7.2 其他函数 54](#_Toc534051483)

[4.8 aodv\_hello.c 54](#_Toc534051484)

[4.8.1 hello\_start()函数-启动hello消息 54](#_Toc534051485)

[4.8.2 hello\_send()函数-发送hello消息 55](#_Toc534051486)

[4.8.3 hello\_process()函数-处理hello消息 57](#_Toc534051487)

[4.8.4 其他函数 58](#_Toc534051488)

[4.9 Nl.c 59](#_Toc534051489)

[4.9.1 Nl\_init函数-初始化函数 59](#_Toc534051490)

[4.9.2 Nl\_kaodv\_callback函数 60](#_Toc534051491)

[4.9.3 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 64](#_Toc534051492)

[4.9.4 Nl\_send函数-发送数据包给内核模块 65](#_Toc534051493)

[4.9.5 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](#_Toc534051494)

[4.9.6 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](#_Toc534051495)

[4.9.7 其他函数 67](#_Toc534051496)

[4.10 Aodv\_neighbor.c 68](#_Toc534051497)

[4.10.1 neighbor\_add函数——更新活跃的邻居节点（更新来自非HELLO AODV控制消息的邻居） 68](#_Toc534051498)

[4.10.2 邻居节点链路中断情况（void NS\_CLASS neighbor\_link\_break） 68](#_Toc534051499)

[4.11 Aodv\_timeout.c 71](#_Toc534051500)

[4.11.1 route\_discovery\_timeout()函数-路由发现超时 71](#_Toc534051501)

[4.11.2 local\_repair\_timeout（）函数-本地修复超时 72](#_Toc534051502)

[4.11.3 hello\_timeout()函数-hello消息超时 73](#_Toc534051503)

[4.11.4 其他函数 73](#_Toc534051504)

[5. AODV仿真 74](#_Toc534051505)

[5.1 仿真思路 74](#_Toc534051506)

[5.2 获取trace文件 74](#_Toc534051507)

[5.2.1 生成仿真环境 74](#_Toc534051508)

[5.2.2 编写aodv.tcl脚本 74](#_Toc534051509)

[5.2.3 执行aodv.tcl脚本 75](#_Toc534051510)

[5.3 分析trace文件 76](#_Toc534051511)

[5.3.1 分组投递率 76](#_Toc534051512)

[5.3.2 路由发起频率 77](#_Toc534051513)

[5.3.3 路由开销 78](#_Toc534051514)

[5.3.4 平均时延 79](#_Toc534051515)

[6. 结束语 80](#_Toc534051516)

# 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。

在实现方面，AODV分为内核模块和用户用户空间两部分，内核模块。主要负责维护一个expl\_entry结构的数组，其中包括目的地，下一条，超时时间等，类似于路由表。用户空间负责接收来自其他节点的消息，更新用户空间维护的路由表，和通知内核更新路由表。需要注意的是，整个AODV协议一共涉及到三个路由表，分别是用户空间的路由表，结构相对复杂，AODV内核模块维护的路由表，结构较为简单，Linux内核维护的路由表。 所以用户空间需要建立两个和内核联系的套接字，分别对应在内核的两个路由表。在3.2中，我们将详细阐述数据包如何在内核模块处理的。

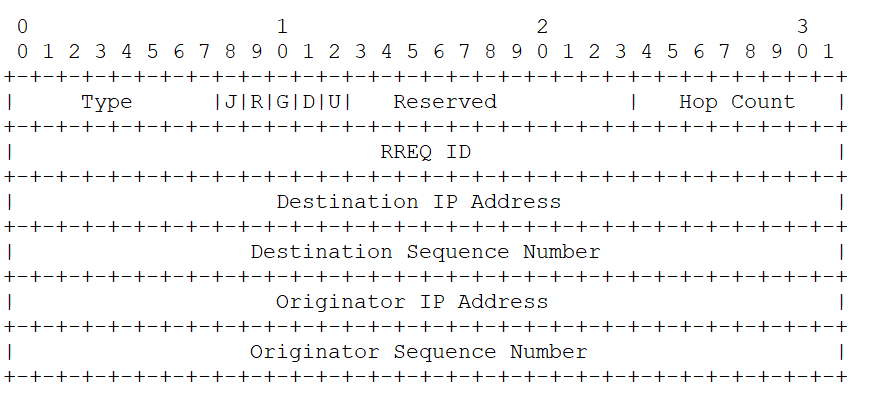
## 特点

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控制帧

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。

### RREQ路由请求帧

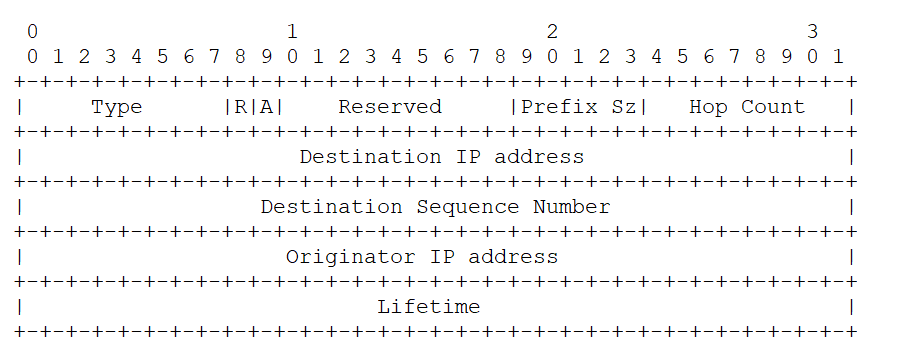


具体每个字段含义如下：

|  |  |
| --- | --- |
| 名称 | 说明 |
| 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; 多播时可能用到 |
| R | 修复标志，多播时可能用到 |
| 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。如果需要转发，就把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

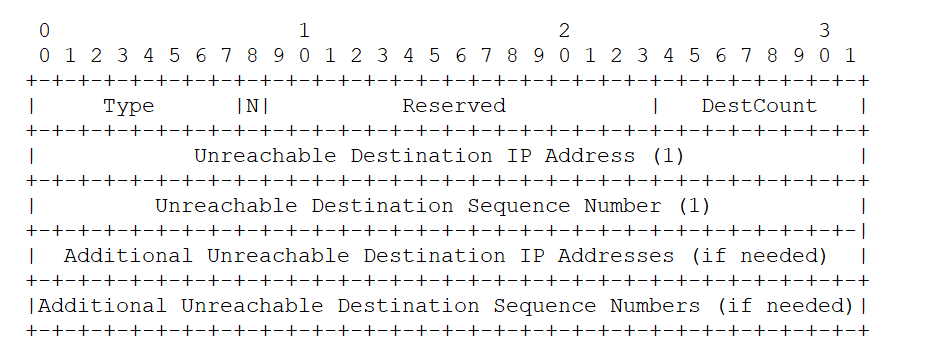


相同的字段与RREQ控制帧意义相同，其余的如下：

|  |  |
| --- | --- |
| 名称 | 说明 |
| 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 |

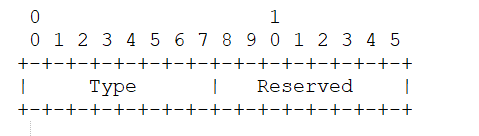
请注意，前缀大小允许子网路由器为由路由前缀定义的子网中的每个主机提供路由，路由前缀由子网路由器的IP地址和前缀大小决定。为了利用这个特性，子网路由器必须保证所有共享指定子网前缀的主机都能访问。详见第7节。当前缀大小不为零时，必须保留与子网路由有关的任何路由信息（和前体数据），而不是该子网上的单个目标IP地址。

### RERR路由错误帧



|  |  |
| --- | --- |
| 名称 | 说明 |
| 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。

## AODV用户态工作流程

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。

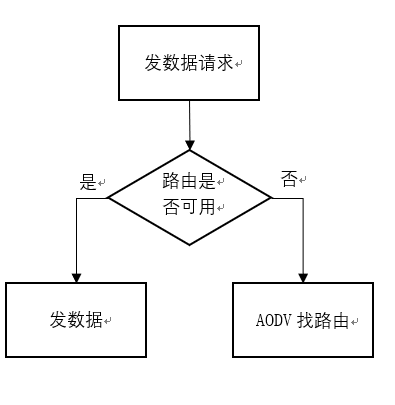


图 1 发现功能触发

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。

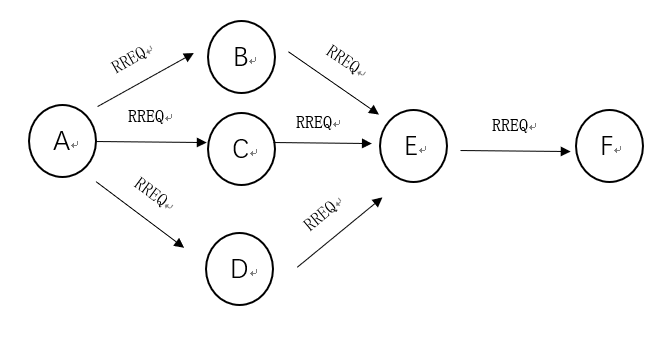


图 2 RREQ帧转发过程

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。

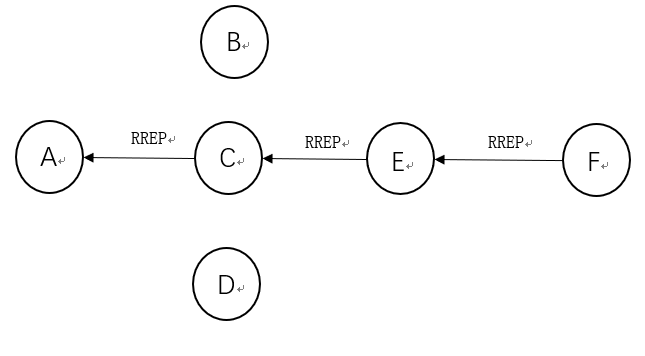


图 3 单播RREP过程

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。

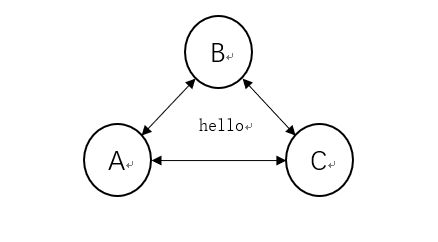


图 4 节点向邻居发hello帧

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。

为了防止不必要的网络范围的rreq，使用扩展环搜索技术。其工作原理如下：源节点连续搜索更大的区域，直到找到目标节点。这是通过对于每个RREQ重传直到找到路由来完成的，增加 每个RREQ分组中携带的生存时间（TTL）值，从而扩展源中心的“搜索环”。

### 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

## 用户空间和内核模块的交互

在实现中，用户空间方面AODV后台进程程序是一个无限循环读取事件队列的进程，首先判断是属于哪一种事件(RREQ，RREP，RERR，HELLO，ACK等) ，再开始启动该事件的处理程序，不断重复读取事件队列，直到返回值是NULL 为止。路由算法的功能通过该进程实现，并作为后台守护进程在用户层执行、负责与其它节点进行信息的交互，建立和维护路由表。AODV 路由的发现和维护是通过发送和接收到控制信息分组经过AODV 算法处理后来完成的，该控制信息由UDP socket发送和接收，并由aodv\_socket 模块进行处理。各类控制信息分组的生成、发送、接收、转发、处理是aodv \_hello、aodv \_rerr、aodv \_rrep、aodv\_rreq等模块协作完成的。

在内核模块，AODV使用了Linux内核中Netfilter的挂接功能，Netfilter为每种网络协议定义了一套钩子函数，这些钩子函数在报文流过协议栈的几个关键点被调用。在这几个关键点上，协议栈将把网络报文，钩子函数以及钩子函数标号作为参数调用。AODV协议用到了定义在IP协议上的钩子函数，通过钩子函数来处理报文，具体将在3.2进行阐述

Netlink socket可以在核心层与用户层之间进行双向的数据交互，当路由后台程序将查找到的路由信息通知给内核以修改核心路由表时，用户层使用标准的socket就可以实现Netlink所提供的强大功能，并且在Netlink 的基础上，使用rtnetlink 可以方便地操作Linux 的核心路由表。当内核控制程序需要通知用户层的后台程序进行路由查找时，核心层需要使用专门的内核API 来实现Netlink 数据交互，Netlink socket将核心路由表的使用状态传递给用户层，告知AODV后台进程核心路由表的使用状况，AODV后台进程据此更新路由缓冲表的定时器，同时通过Netlink socket 删除内核路由表中过时的路由条目或增加新的路由表项。具体实现结构图如下：



**图 5 AODV实现结构**

# 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 |
| Kaodv\_mod | 内核主程序，初始化和注销内核 |
| Kaodv\_netlink | 内核层与用户层进行通信 |
| Kaodv\_debug | 显示核心层线程处理流程信息 |
| Kaodv\_ipenc | 对内核路由表中IP地址编码 |
| Kaodv\_queue | 将数据包放入内核队列 |
| Kaodv\_expl | 内核路由表信息到期列表 |

# 内核空间函数解读

## Linux钩子函数简介

在AODV协议的具体实现过程中，Netfilter在处理数据包流程时起到关键性的作用，Netfilter由处于Linux 协议栈中不同点上的五个钩子(hook)关键点组成，用户可以在这些位置注册自己定义的操作函数，经过hook点的数据分组将执行函数操作，帮助用户完成数据分组的过滤和修改等功能。如图所示



数据包网络入口以后，首先对IP进行校验后，经过注册在NF\_ IP\_PIFORWARD的钩子函数处理后，进入内核路由处理模块，接着判断数据包是发送给本机的还是需要转发的，如果是发给本机的，则该数据包经过注册在NF\_IP\_LOCAL\_IN的钩子函数传递给上层协议，若该数据包应该被转发则把它被交给注册在NF\_ IP\_PIFORWARD的钩子函数处理，转发的数据包经过注册在NF\_IP\_LOCAL\_OUT的钩子函数处理后，再发送到网络上。本地产生的数据包经过注册在在NF\_IP\_LOCAL\_OUT钩子函数处理后，进行路由选择处理，然后经注册在NF\_IP\_POST\_ROUTNIG钩子函数处理，最终发送到网络上。每个注册的钩子函数经过处理后都将通过返回值告知Netfilter核心代码，以便对数据包采取相应的动作。返回值的功能如下: NF\_ACCEPT: 继续正常的数据包处理; NF\_DROP: 将数据包丢弃; NF\_STOLEN: 由钩子函数处理了该数据包，不再继续传送; NF \_QUEUE: 将数据包入队，通常交由用户程序理;NF\_REPEAT: 再次调用该钩子函数。

## 内核空间对数据包处理流程

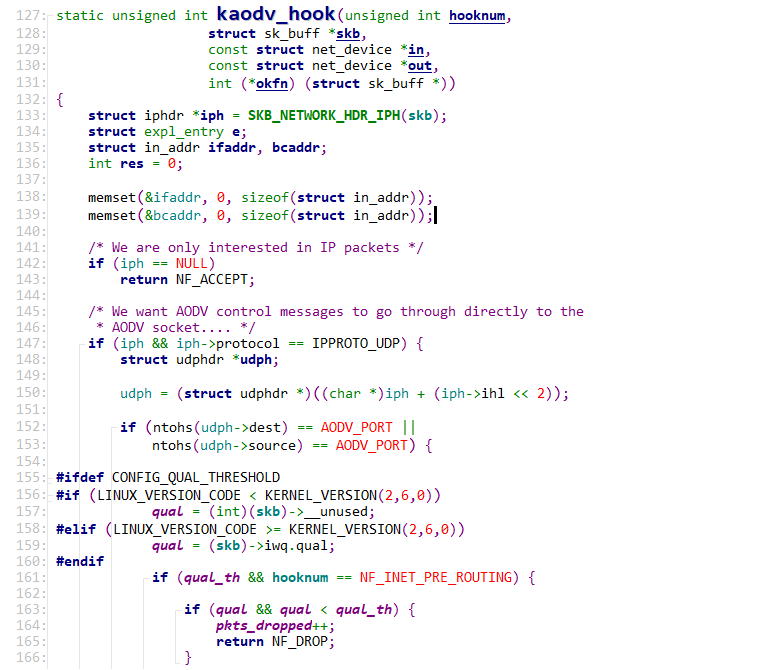
本协议用到了五个关键点中的三个分别为F\_IP\_PRE\_ROUTING(流入数据包在决策路由前执行) 、NF\_IP\_LOCAL\_OUT(本地产生数据包流出路由前执行) 、NF\_IP\_POST\_ROUTING( 在本地数据包或转发数据包在发送之前) 。

在本地直接发送或者转发数据包时，首先判断内核路由表中是否存在与该数据包的目的地址相匹配的路由条目，如果存在，根据该路由条目信息提供的相应网络接口将其发送到对应的下一跳节点地址上，如果不存在与数据包目的地址相匹配的路由条目，向用户层的后台进程发起路由请求，由后台进程发起路由查找过程，并且在路由查找结束和内核路由表更新之前，数据包通过挂接在NF\_IP\_LOCAL\_OUT hook 点的回调函数进行处理，并通过用Linux的原始套接口( Raw Socket) 将数据包送往用户层的数据缓冲区中进行排队。同时，必须用户层维护一个与内核路由表形成映射关系的路由缓冲表。路由缓冲表的所有路由条目都有一个与之相关的定时器，当内核路由表中的路由条目被使用时，与之对应的用户层缓冲路由表的定时器必须被重置，当定时器的时间到期时，该表项就要从路由缓冲表和内核路由表中同时删除。如果查找到与被缓存数据包目的地址匹配的路由，则将这条路由插入到内核路由表中，用原始套接口将缓存的数据分组重新发送出去。如果在规定的时间内未找到相应的路由，则这些缓存的数据报文会被丢弃，并通知源节点出错。

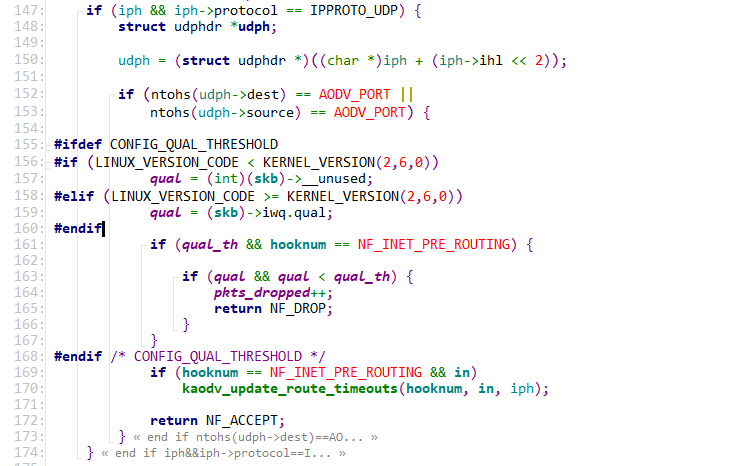
## kaodv-mod.c

本文件是内核模块开始的文件，主要包括了要在三个关键点上挂载的kaodv\_hook函数和更新AODV内核模块维护的路由表的函数

### 3.3.1 kaodv\_hook-aodv-钩子函数

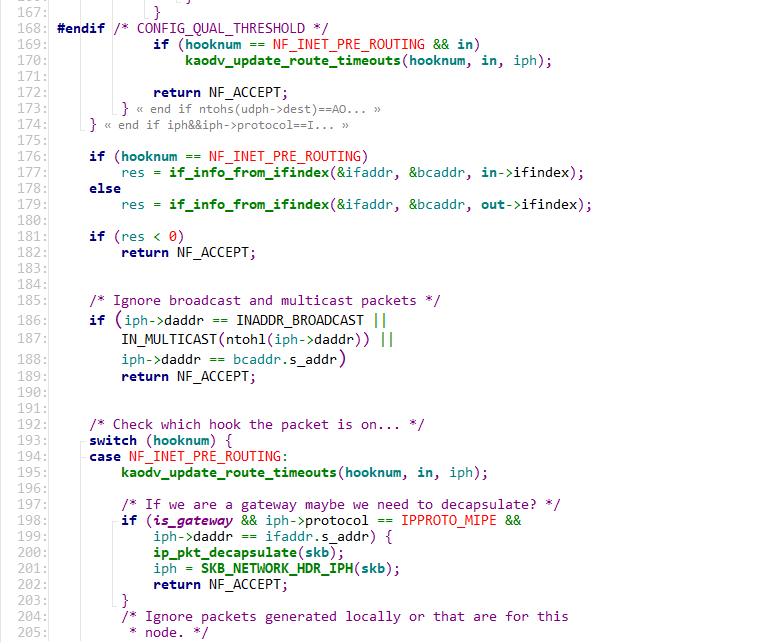


142-143：如果IP为空，则继续正常的数据包处理，即忽略这个数据包，因为协议只对IP数据包感兴趣



147-153：检测是否是AODV socket发来的消息

162-165：如果消息过时了，则丢弃它

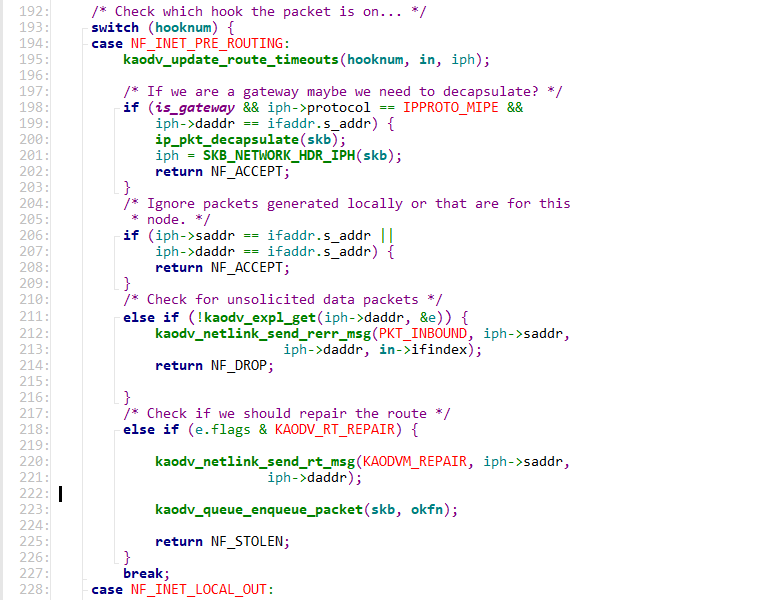


176-177：如果流入钩子函数标号代表数据包在决策路由前则查找输入的端口信息，若找到则res=0；否则res=-1

176-177：如果流入钩子函数标号代表本地生成数据包或者转发数据包时则查找输出的端口信息，若找到则res=0；否则res=-1

181-182：若果没有找到端口信息，则忽略该数据包

186-189：如果是广播或者多播信息，则忽略该数据包



193-193：查找数据包是哪个钩子函数钩来的

194-194：如果是NF\_INET\_PRE\_ROUTING，即流入数据包在决策路由前钩来的

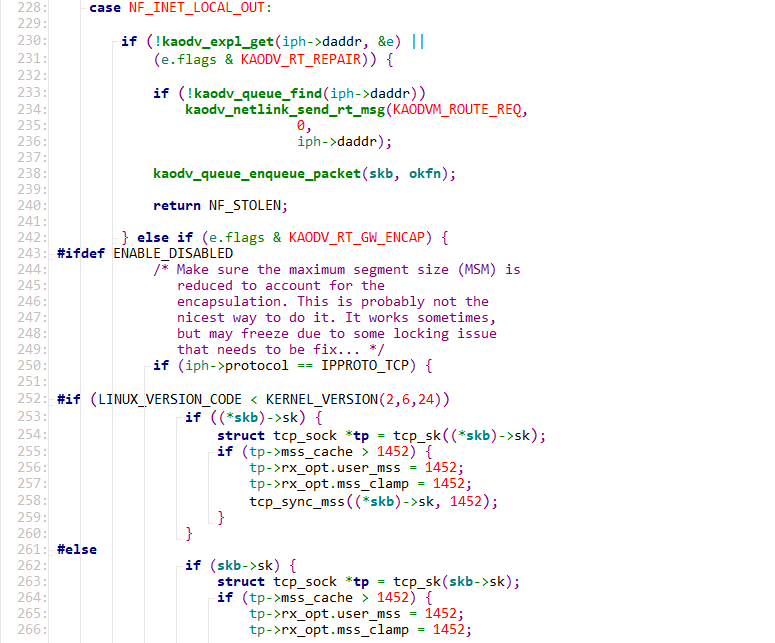
195-195： 更新路由过期时间包括更新expl\_entry结构和通知用户进程更新用户空间的维护的路由表

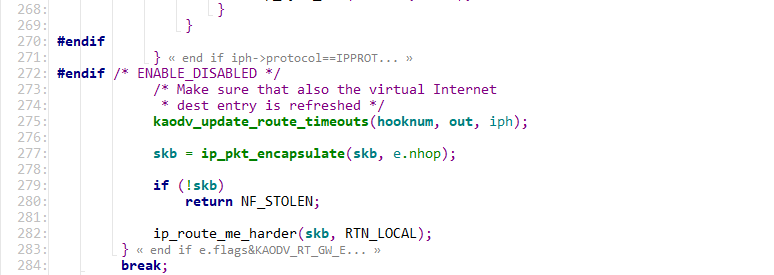
198-203：如果是网关的话，则将数据包解封装

206-209：忽略忽略本地生成的或为此节点生成的数据包

211-216：如果本地路由表中没有数据包的中目标地址的信息，则发送rerr消息，然后丢弃数据包

218-226：如果找到了目标节点的路由，判断是否需要flags是否要修复，如果需要，向用户空间发送修复信息，并将当前数据包放进数据缓存队列中





228-228： 如果是NF\_INET\_LOCAL\_OUT，即本地产生数据包流出路由前钩来到

230-240：如果不存在到目的节点的路由，或者路由需要修复的情况下，首先判断数据缓存队列中有没有关于该目的节点的数据packet,如果有的话，表示已经发送RREQ进行路由请求了，如果没有，则向用户空间通信，发起RREQ路由请求，同时该缓存数据packet进入缓存队列中，不再发送该数据包

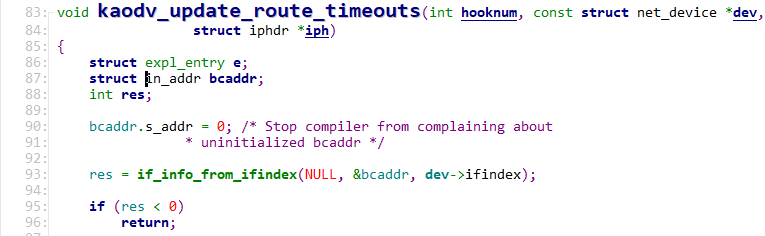
242-269：如果路由条目有KAODV\_RT\_GW\_ENCAP,且上层协议为TCP，设置最大报文段长度为1452B，超过才长度则分片

275-283：更新expl\_entry结构和通知用户进程更新用户空间的维护的路由表，将数据包打包



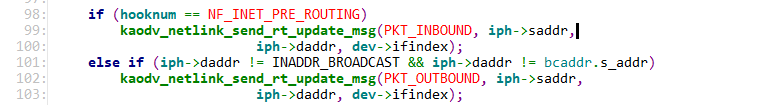
如果是NF\_INET\_POST\_ROUTING，则更新expl\_entry结构和通知用户进程更新用户空间的维护的路由表

### 3.3.2 kaodv\_update\_route\_timeouts-更新expl\_entry结构和通知用户进程更新用户空间的维护的路由表

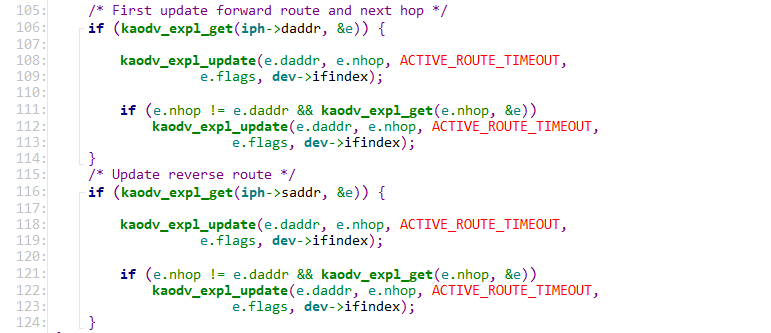


093-093：根据ifindex查找端口信息

095-096：如果没有找到则退出

098-100：如果数据包是在流入方向被钩到的，则想用户空间发送流入方向路由更新

101-103：如果数据包是在流出方向被钩到的且不是广播数据包，则想用户空间发送流出方向路由更新



105-114：更新expl\_entry结构的前向路由和下一跳

116-124：更新反向路由

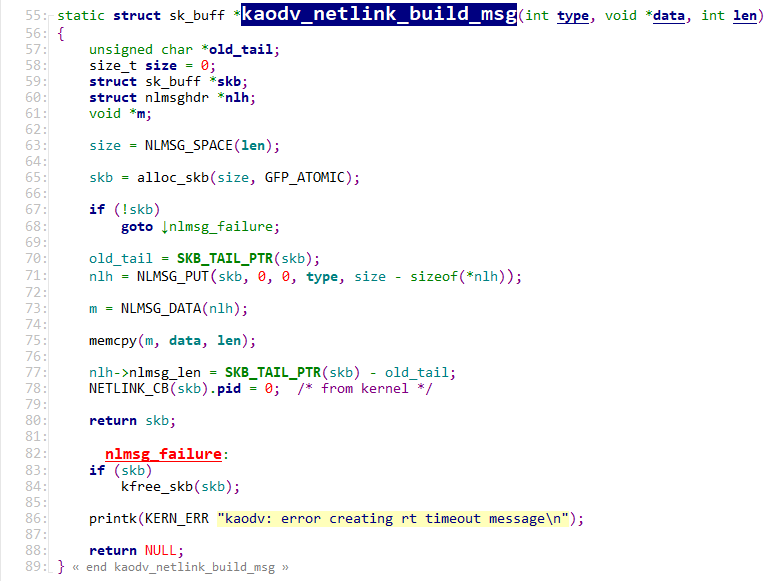
### 3.3.3 其他函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| kaodv\_init | 初始化缓冲队列，初始化设备端口信息等 |
| kaodv\_exit | 退出AODV函数，注销钩子函数，释放缓冲队列，路由表等空间 |

## kaodv-netlink.c

本文件实现了关于AODV协议netlink机制相关的函数

### 3.4.1kaodv\_netlink\_build\_msg-创建sk\_buff结构体



059-059： 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

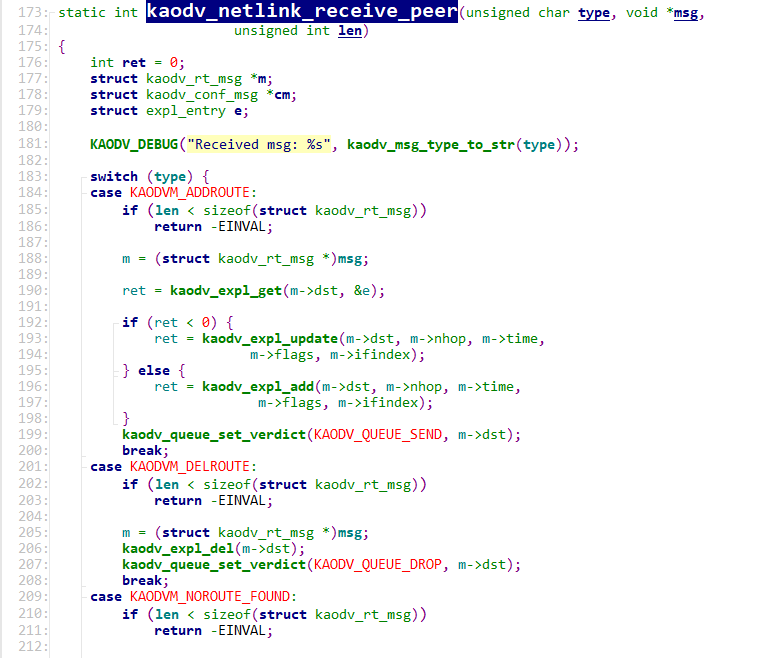
060-060：nlh是一个nlmsghdr类型的指针

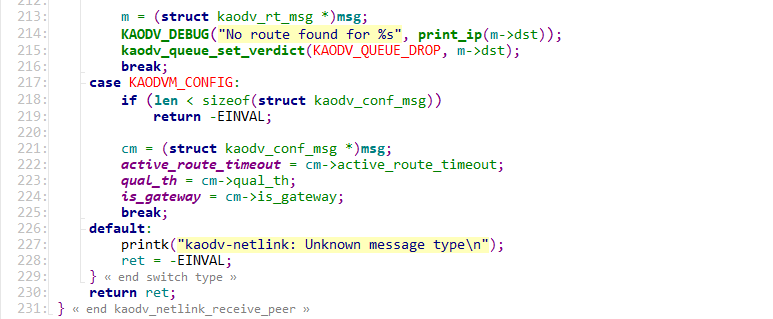
065-065：alloc\_skb函数分配缓冲区，包括两个缓冲区，一个是数据缓冲区，一个是缓冲区的描述结构sk\_buff

067-088：如果alloc\_skb失败，则释放skb内存并返回null

070-080：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

### 3.4.2 kaodv\_netlink\_receive\_peer-内核收到消息后的处理的方式





177-177：m是一个kaodv\_rt\_msg的指针，携带有路由信息

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

179-179：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

### 3.4.3其他函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| kaodv\_netlink\_send\_debug\_msg | 输出调试信息 |
| kaodv\_netlink\_send\_rt\_msg | 向应用空间广播路由信息 |
| kaodv\_netlink\_send\_rt\_update\_msg | 向应用空间广播路由更新信息 |
| kaodv\_netlink\_send\_rerr\_msg | 向应用空间广播端口路由信息 |
| kaodv\_netlink\_rcv\_nl\_event | 如果是内核消息，则刷新expl\_entry和缓冲队列 |
| kaodv\_netlink\_rcv\_skb | 处理sk\_buff，调用kaodv\_netlink\_receive\_peer处理消息 |
| kaodv\_netlink\_init | Netlink初始化，注册事件通知链 |
| kaodv\_netlink\_fini | 注销事件通知链 |

## kaodv-expl.c

本文件负责维护expl\_entry结构体，即内核中的路由表，该结构体包括过期时间，标准，目的地IP，跳数，端口编号信息

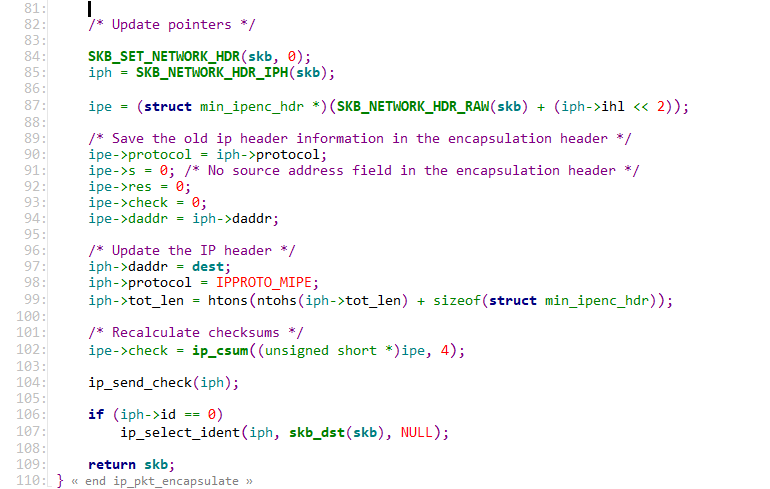
|  |  |
| --- | --- |
| 函数名 | 说明 |
| kaodv\_expl\_timeout | 获取当前时间，将已经失效的路由表项删除，注意对路由表的操作是加写锁的 |
| \_\_kaodv\_expl\_flush | 删除路由表 |
| \_\_kaodv\_expl\_add，kaodv\_expl\_add | 按过期时间顺序增加路由表项 |
| \_\_kaodv\_expl\_find | 根据IP地址查找特定的路由表项 |
| \_\_kaodv\_expl\_del | 根据路由表项删除特定路由表项 |
| kaodv\_expl\_del | 根据IP地址删除特定的路由表项 |
| kaodv\_expl\_get | 查找特定路由表项，若查到返回1，否则返回0 |
| kaodv\_expl\_print | 打印expl\_entry结构体 |
| kaodv\_expl\_proc\_info | 返回剩下的分片长度 |
| kaodv\_expl\_update | 更新expl\_entry结构体，更新时加写锁 |
| kaodv\_expl\_init | 初始化expl\_entry结构体 |
| kaodv\_expl\_fini | 删除expl\_entry结构体 |

## kaodv-ipen.c

本文件负责在两个不直接相连的AODV网络传递信息时，在边缘网关将IP数据包封装，使其能在普通网络中传递，类似于IPv6的数据包的隧道技术。

### ip\_pkt\_encapsulate-IP数据包封装





54-63：在首部申请新的空间，若果申请失败打印失败信息并退出

65-71：第67行将nskb这个数据缓冲区控制结构和数据缓冲区关联起来。第71行修改指向数据区末尾的指针，并向下扩大ipenc头部大小

073-077：将IP头部和数据都后移

079-079：释放原来的sk\_buff

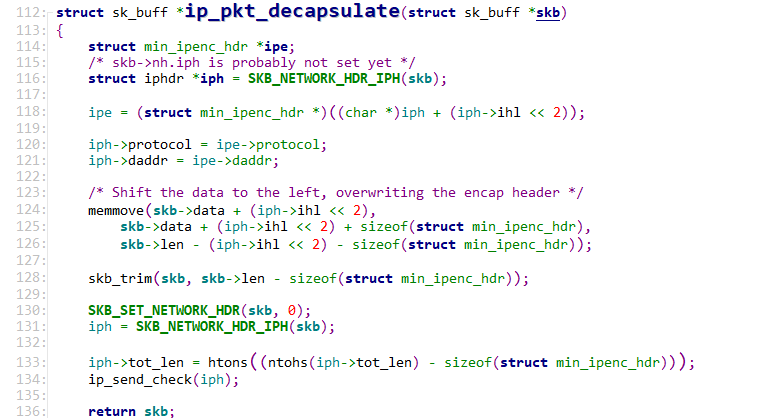
084-087：设置首部指针

090-094：将原来的IP首部信息保存到封装的首部

097-099：更新IP头部，将协议类型改为自己定义的IPPROTO\_MIPE,重新计算包大小

102-102：重新计算首部校验和

### ip\_pkt\_dncapsulate-IP数据包解封装



124-126：由于封装时讲原来的首部和数据右移，只需要将其左移覆盖封装的首部即可

133-134：重新计算数据包长度并检验

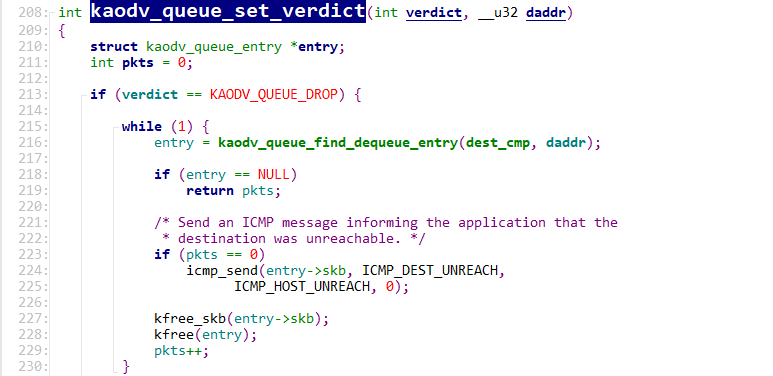
### 其他函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| ip\_pkt\_encapsulate | 计算IP首部校验和 |

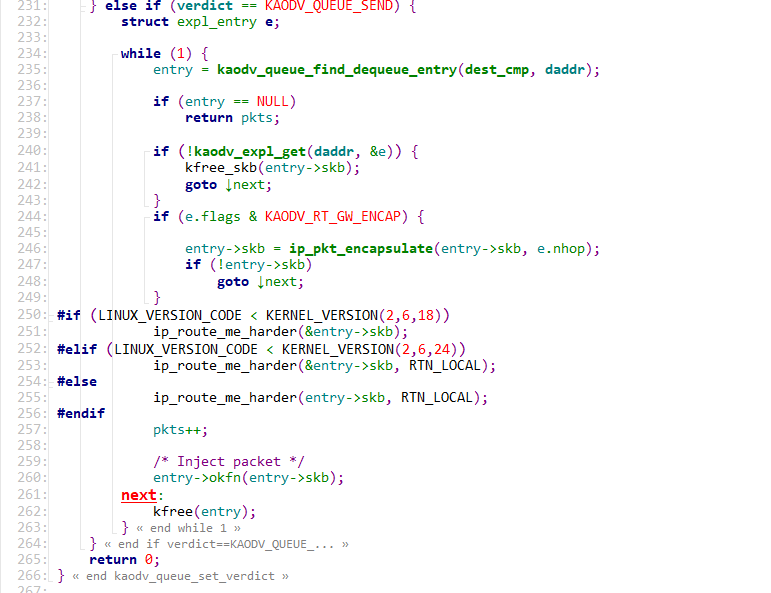
## kaodv-queue.c

本文件负责建立数据包缓存队列，注意对于该队列的写操作都是加写锁。同时该文件还接受来自AODV内核模块的消息处理队列里的数据包即kaodv\_queue\_set\_verdict实现的功能。

### kaodv\_queue\_set\_verdict-队列裁决函数



213-230：如果是需要丢弃的，查找队列，找到所有目的地为该IP地址的数据包做出队操作，并且发送ICMP消息告知目的地不可达



234-263：如果是需要发送的，找到所有目的地为该IP地址的数据包，如果是网关，则讲数据包封装后交给kofn指向的函数处理，如果不是网关，直接交给okfn指向的函数处理

|  |  |
| --- | --- |
| 函数名 | 说明 |
| kaodv\_queue\_enqueue\_packet | 元素入队，其中队列元素的属性其中okfn是该数据包经过所有钩子函数后需要执行的函数指针 |
| kaodv\_queue\_find\_dequeue\_entry | 删除队列元素 |
| kaodv\_queue\_flush | 清空队列 |
| kaodv\_queue\_find | 查找队列中是否对应IP的元素 |
| init\_or\_cleanup | 初始化队列或者删除队列，如果是0则清空并删除队列 |

# 用户态函数解读

## 用户态全局变量

### 路由表项

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。

### 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 | 删除某位置上的节点 |
| static inline int listelm\_add | 在指定位置添加节点 |
| int list\_add | 在头部添加一个节点 |
| int list\_add\_tail | 在末尾添加一个节点 |
| int list\_detach | 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 | 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 | 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 | 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。

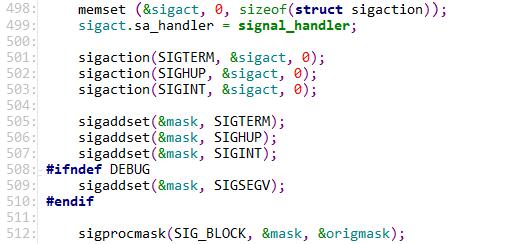
|  |  |
| --- | --- |
| 名称 | 功能 |
| int NS\_CLASS timer\_init | 定时器初始化 |
| void NS\_CLASS timer\_timeout | 定时器超时时进行清理工作 |
| NS\_STATIC void NS\_CLASS timer\_add | 在定时器队列中加一个定时器 |
| int NS\_CLASS timer\_remove | 从定时器队列中移除一个定时器 |
| int NS\_CLASS timer\_timeout\_now | 手动让一个定时器超时，以触发超时事件 |
| void NS\_CLASS timer\_set\_timeout | 给指定定时器增加一定的时间 |
| long timer\_left | 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 | 打印出每个定时器的信息 |

## main函数

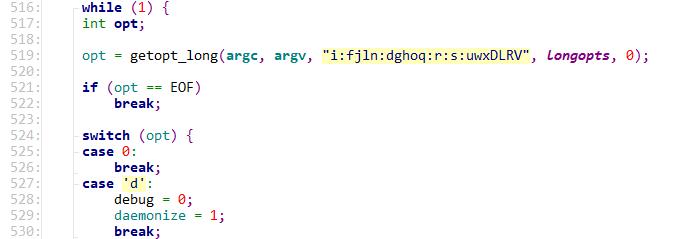
main函数主要功能是对于各个变量的初始化，包括应用到的与内核沟通的套接字，与其他节点通信的套接字，全局链表，路由表，信号等。读取命令行参数以对相关变量进行设置，使得协议执行不同的动作。最终通过pselect轮询函数触发回调函数来实现节点的各个功能。



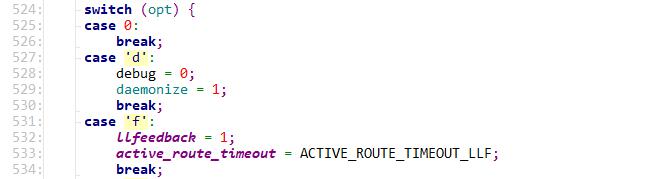
481-496：声明各变量，包括文件描述符集，计时器，信号量，读取文件名等



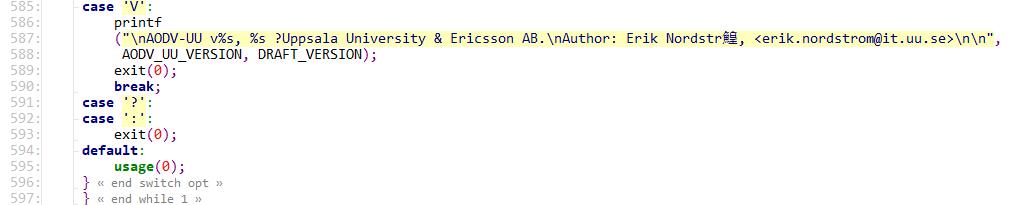
498-512：先是通过memset函数来清空结构体的内存，然后对信号结构体进行相关设置，最后将信号结构体阻塞住，将来在pselect函数中处理它。



519-522：永真循环中用来提取命令行参数，紧接着判断如果是EOF结束符则跳出这个永真循环

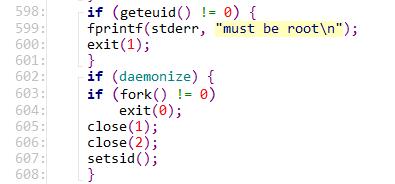


/\* 期间结构是极其相似的，都是case+命令符。执行对应的操作\*/



515-597：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 |



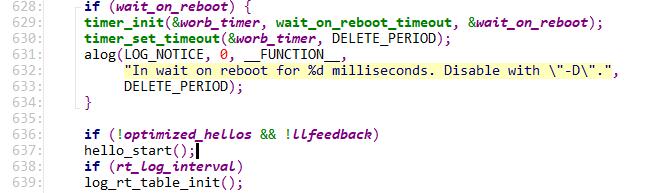
598-601 ：检查AODV是否在root权限下运行，否则退出

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

636-637：启动HELLO帧，准备发第一个

638-639：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。

640-662 行代码所示的永真循环为main函数的核心。函数前部分通过初始化设定好触发事件，比如内核消息的传递，控制帧的接收，路由表项的更新，以及定时器超时等等。当触发事件发生调用相应的回调函数来处理事件，完成AODV的功能。

## main函数文件内其他函数以及变量的定义：

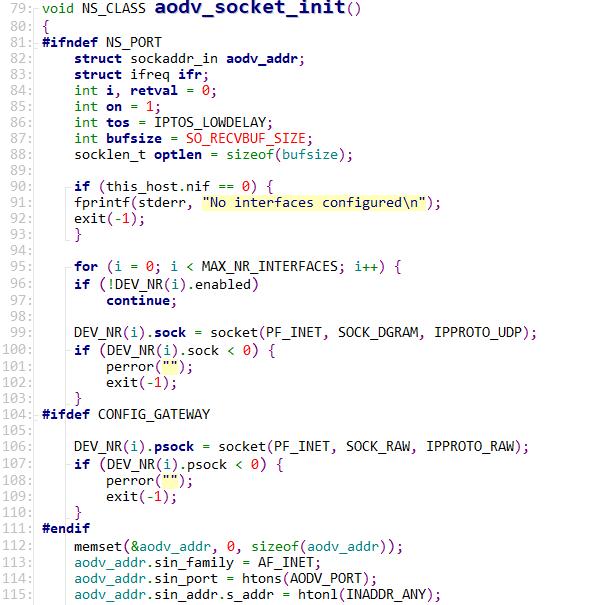
在这里以表格的形式简单说明main.c中剩下的函数功能和变量

|  |  |
| --- | --- |
| 名称 | 说明 |
| voidusage | 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 | 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 | 寻找节点默认网关 |
| struct sockaddr\_in \*get\_if\_info | 根据套接字接口名字和其类型获取详细信息 |
| void host\_init | 初始化某个端口 |
| void load\_modules | 装载内核模块 |
| void remove\_modules | 删除某个模块 |
| int set\_kernel\_options | 设置内核选项 |
| void signal\_handler | 信号处理函数，根据不同类型的信号作不同的操作 |

## AODV\_socket.c

### 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。



079-088：声明UDP套接字所需要的变量

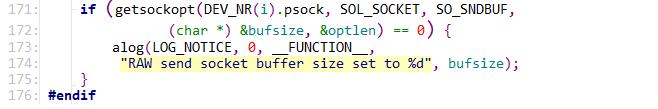
090-093：如果没有可用接口的话之间退出

095-097：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。

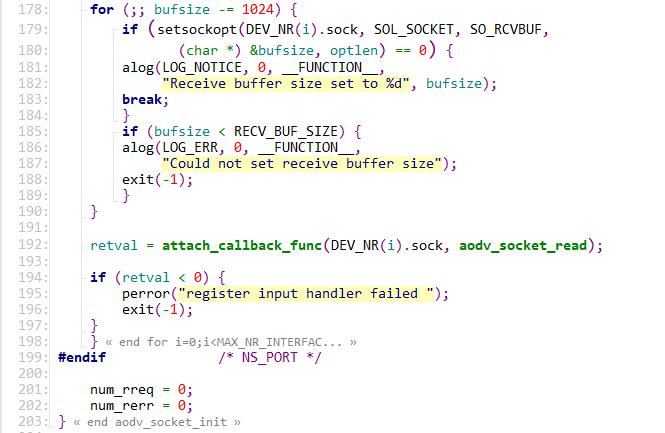
099-115：创建并检测生成的UDP套接字和与内核沟通的原始套接字，设置aodv地址参数



/\*设置套接字选项\*/



120-176：设置套接字选项。省略部分代码结构和124-128部分一致，每一个if语句包含了一个套接字选项的设置。用这样的代码设置了套接字选项：SO\_PRIORITY，SO\_BINDTODEVICE，SO\_BROADCAST，IP\_PKTINFO，IP\_RECVTTL，SO\_BINDTODEVICE，SO\_SNDBUF，SO\_SNDBUF。如果其中一个设定失败则显示相应的错误信息，并返回

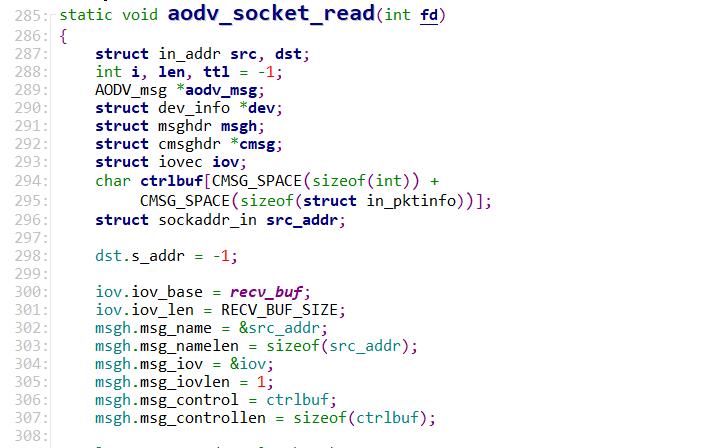


178-190：设置收控制帧的缓冲区大小，失败则显示消息并返回

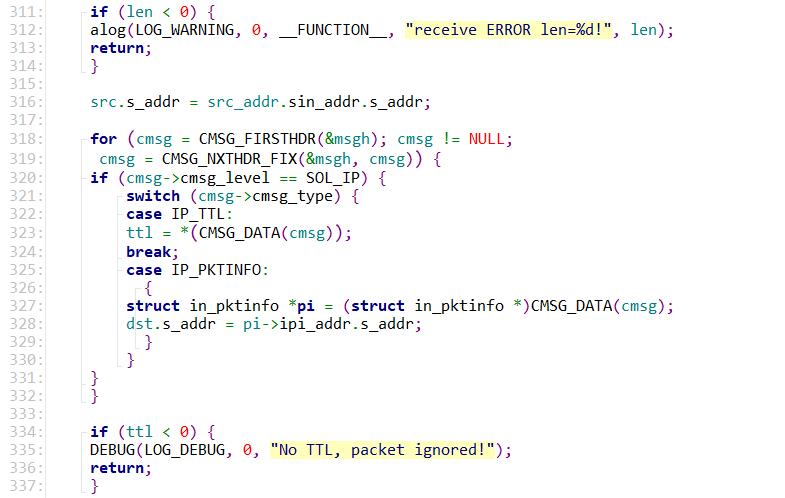
192-192：把创建好的套接字挂载到主函数的callback结构体当中

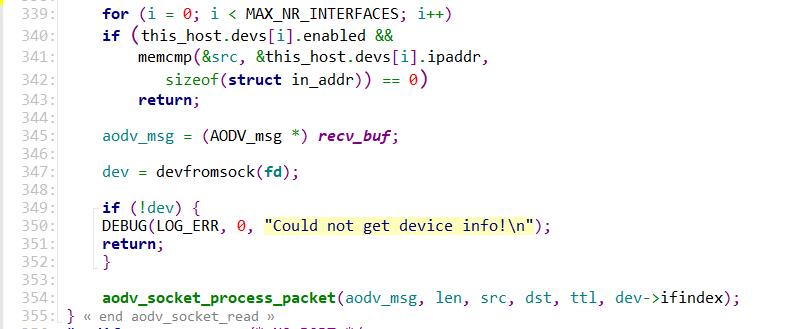
### aodv\_socket\_read函数-套接字读取

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。



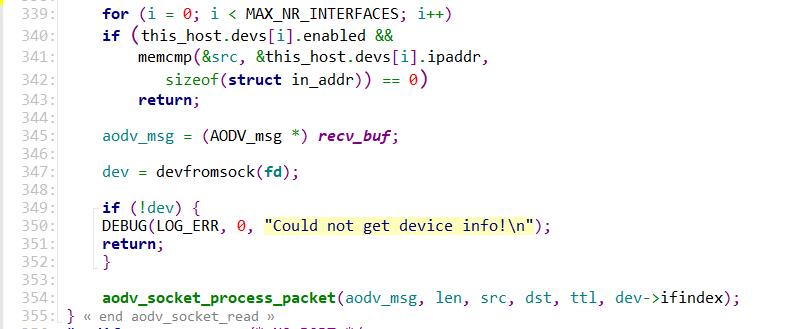
287-307：声明各消息变量，提取消息，以待使用，





309-314：收取套接字接收到的信息，如果长度小于0，则丢弃

334-343：判断如果ttl<0或者是本节点生成的数据包则忽略掉。



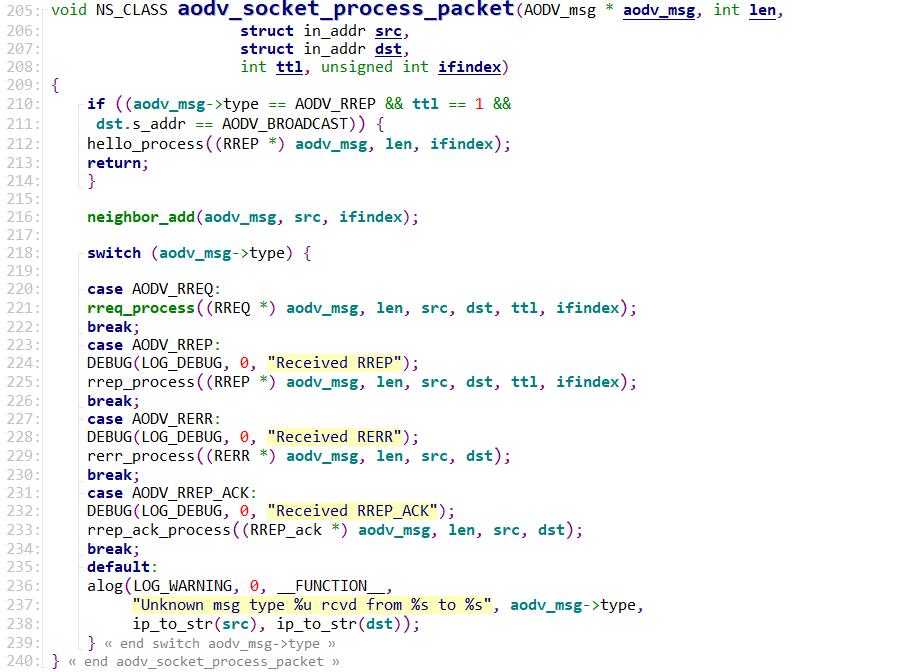
345-345：提取相应消息

347-352：提取设备的信息，如果没有成功，则DEBUG相应的信息

354-354：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。

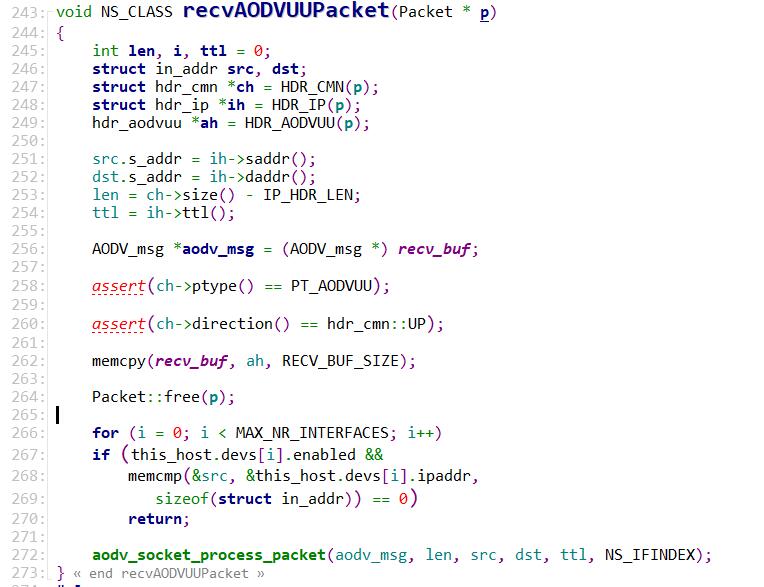


210-214： 如果包是一个HELLO帧，则把该包传给hello\_process函数进行处理

216-216： 根据aodv消息，src和ifindex参数，节点添加邻居节点

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

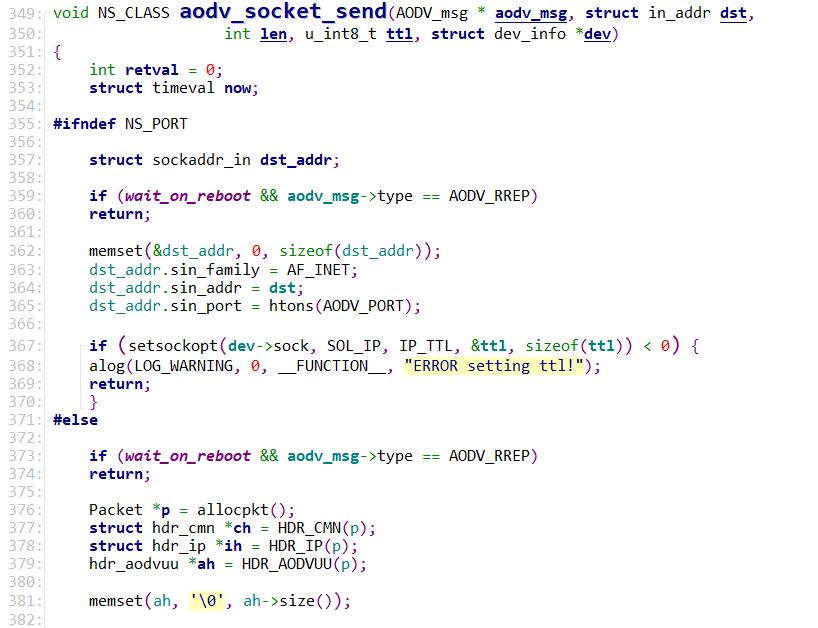


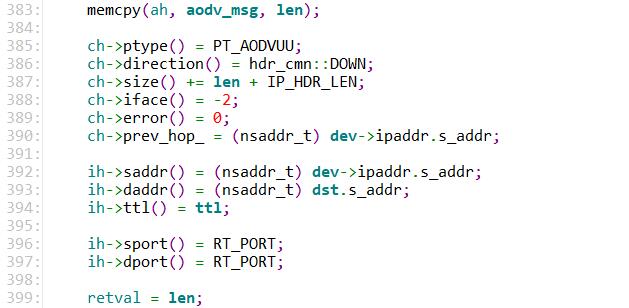
245-264： 给收到的数据包先分配缓存空间，然后释放掉

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

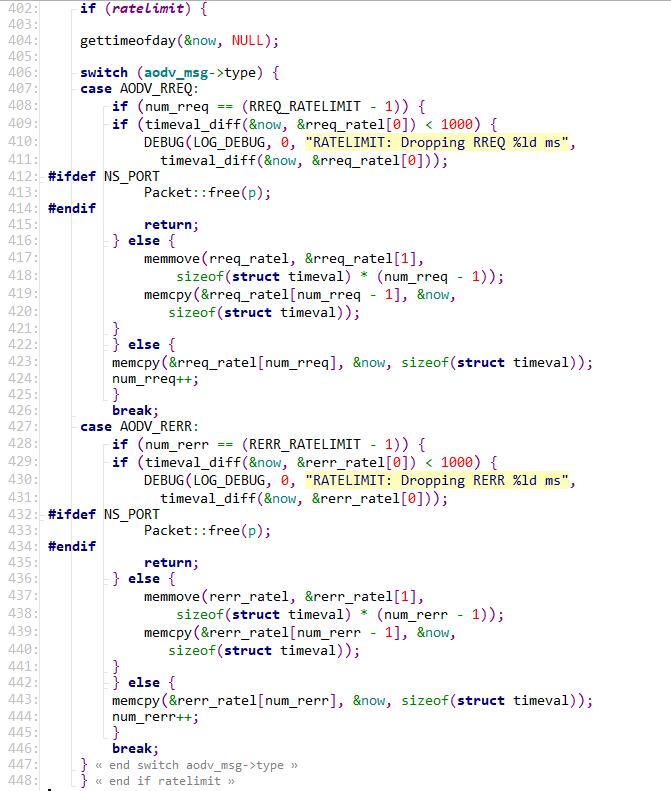
272-272： 把包递交给下一函数进行处理

### aodv\_socket\_send函数-套接字发送函数

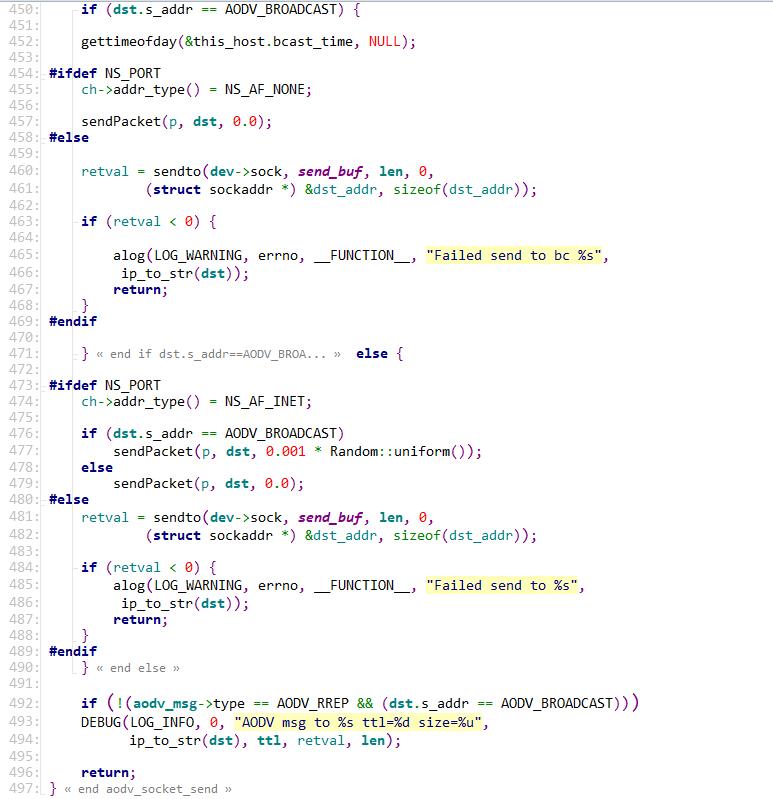




352-399：如果没有定义NS\_PORT设置头部消息和TTL，否则检查是否是RREP，不是则为包申请空间，将相关信息填入。



402-448：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。



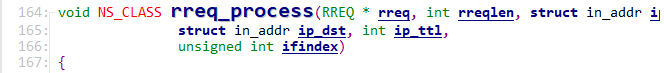
450-497：如果要发送的包是要广播的，那么更新最后广播该数据的时间，能够避免HELLO消息重复广播。

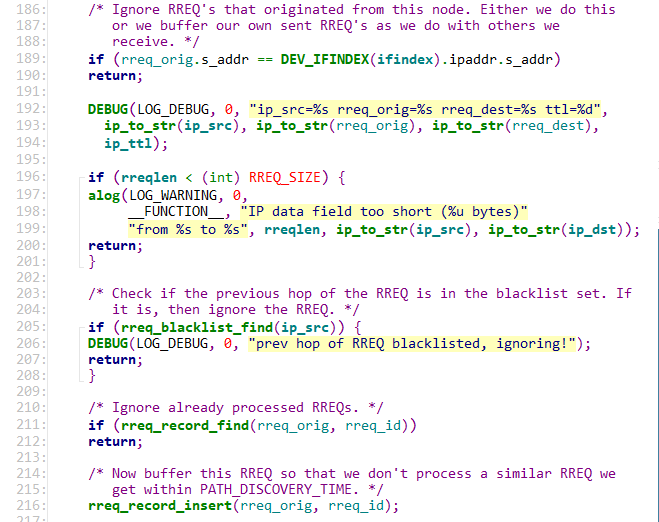
### 其他函数

|  |  |
| --- | --- |
| 名称 | 说明 |
| aodv\_socket\_new\_message | 为新消息分配空间，并返回该消息的指针 |
| aodv\_socket\_queue\_msg | 一个AODV消息存储在发送队列中 |
| aodv\_socket\_cleanup | 关闭每个开放接口的套接字连接 |

## AODV\_rreq.c

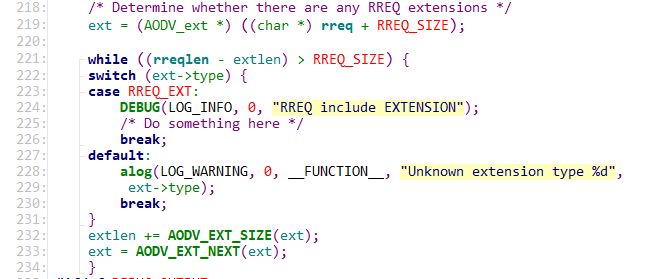
### rreq\_process函数-处理rreq消息



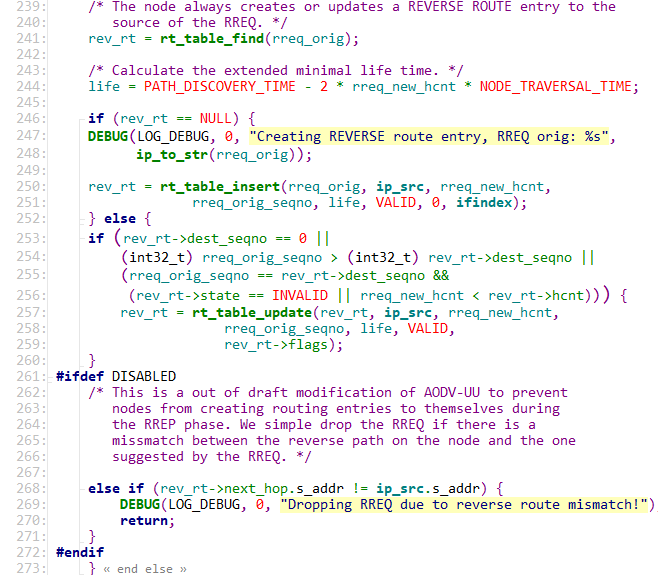


186-212：检查rreq消息，如果来自节点自身或已被加入黑名单或已有记录，则忽略该消息

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



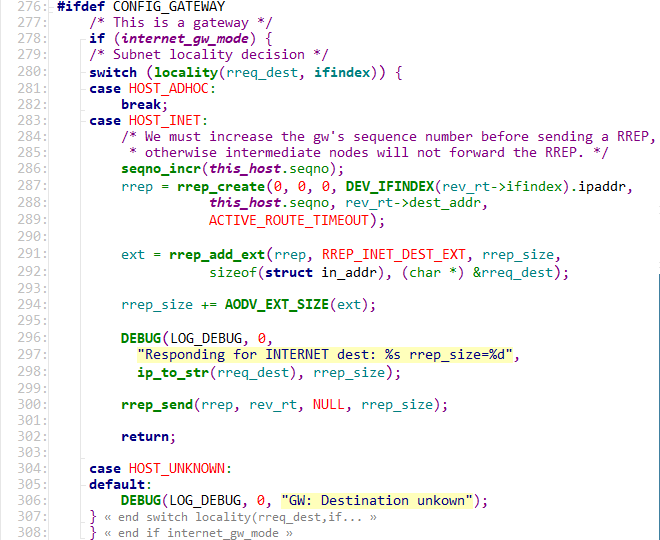
218-237：检查是否还存在RREQ拓展消息



241-273：更新反向路由，具体方法为：

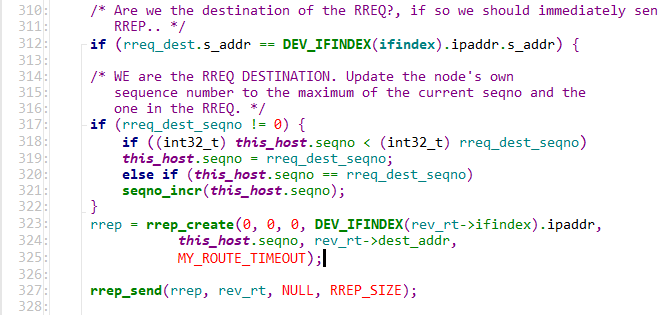
1. 如果路由表没有该rreq的源地址，则插入一条关于该源地址的路由表项；
2. 如果路由表有关于源地址的路由表项，满足下面三个条件之一则更新路由表项
3. 路由表的目的地址序列号为0；
4. Rreq的目的地址的序列号是否大于路由表项的序列号；
5. 序列号相等但是rreq跳数较少或者该路由表项因超时等失效了

268-271：检查是否在RREP阶段建立了自环

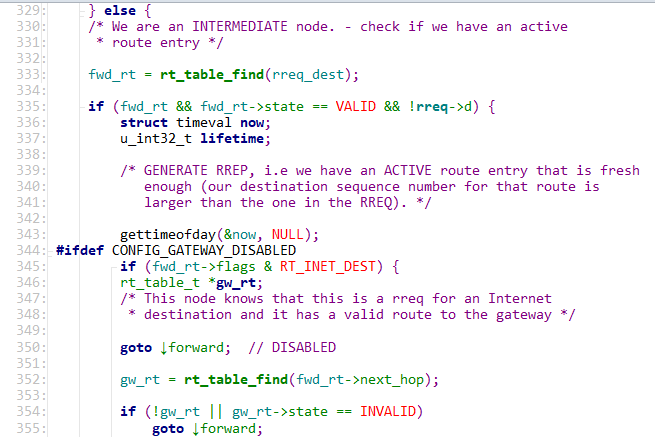


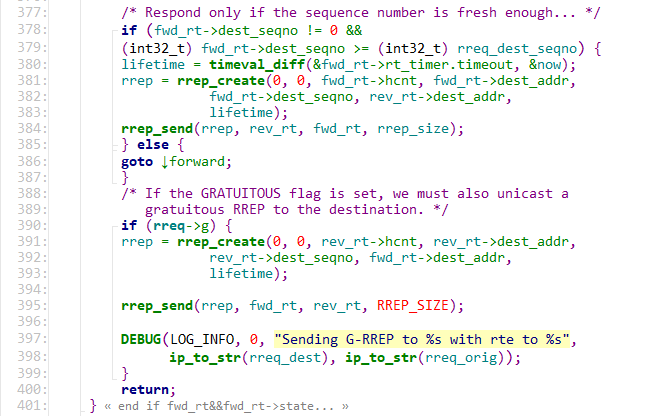
281-306：如果网关的模式，如果为HOST\_INET，则序列号自增，并创建和发送rrep消息

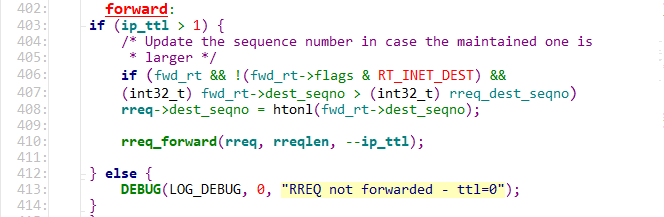
防止中间节点不转发rrep消息？



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

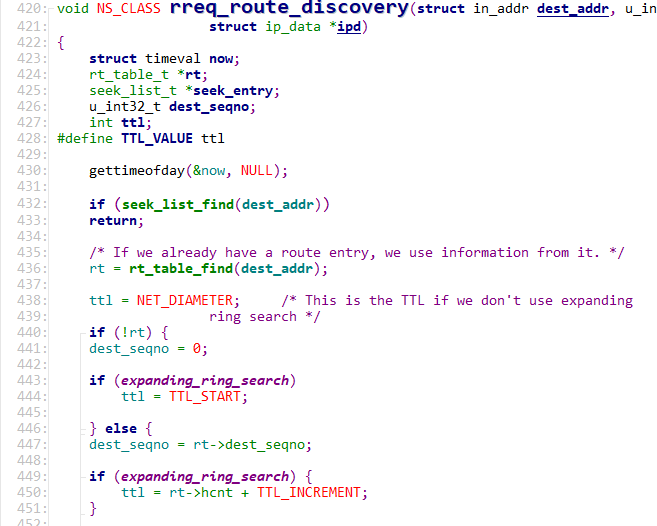


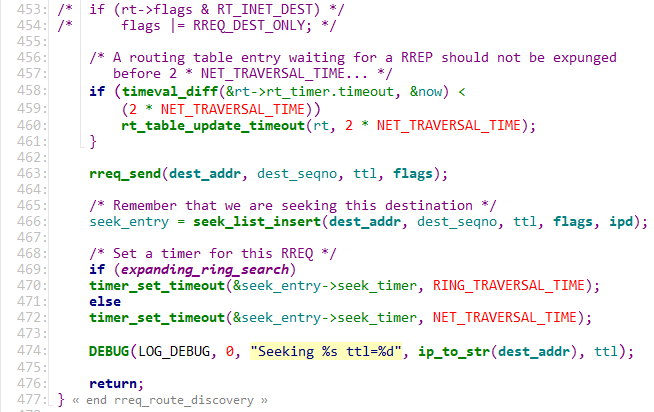




329-414：如果本节点不是目标节点则查找路由表中有无该路由表项如果有有效的路由表项，当路由表的序列号比rreq目的地序列号大或者rreq的g为1的时候回复rrep否则转发ttl>1的rreq消息

### rreq\_route\_discovery函数-路由发现过程





432-432：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

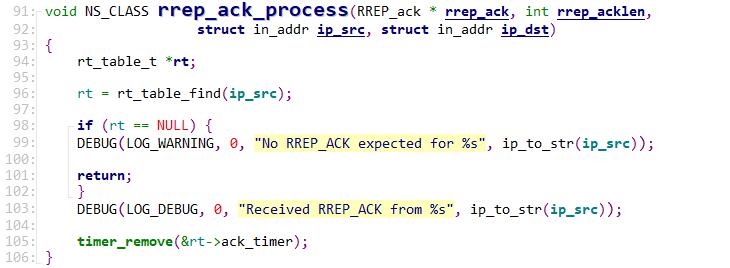
### 其他函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| void NS\_CLASS rreq\_send | 向所有端口转发rreq消息 |
| void NS\_CLASS rreq\_forward | 跳数加1，并向所有端口转发rreq消息 |
| RREQ \*NS\_CLASS rreq\_create | 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 | 增加rreq拓展消息 |
| void NS\_CLASS rreq\_local\_repair | 作用与路由发现函数类似 |
| NS\_STATIC struct rreq\_record \*NS\_CLASS rreq  \_record\_insert | 记录rreq消息 |
| NS\_STATIC struct rreq\_record \*NS\_CLASS rreq\_record\_find | 查找rreq消息是否在记录中 |
| void NS\_CLASS rreq\_record\_timeout | 销毁记录 |
| struct blacklist \*NS\_CLASS rreq\_blacklist\_insert | 将某目的地址加入黑名单 |
| struct blacklist \*NS\_CLASS rreq\_blacklist\_find | 查找目的地址是否在黑名单中 |
| void NS\_CLASS rreq\_blacklist\_timeout | 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

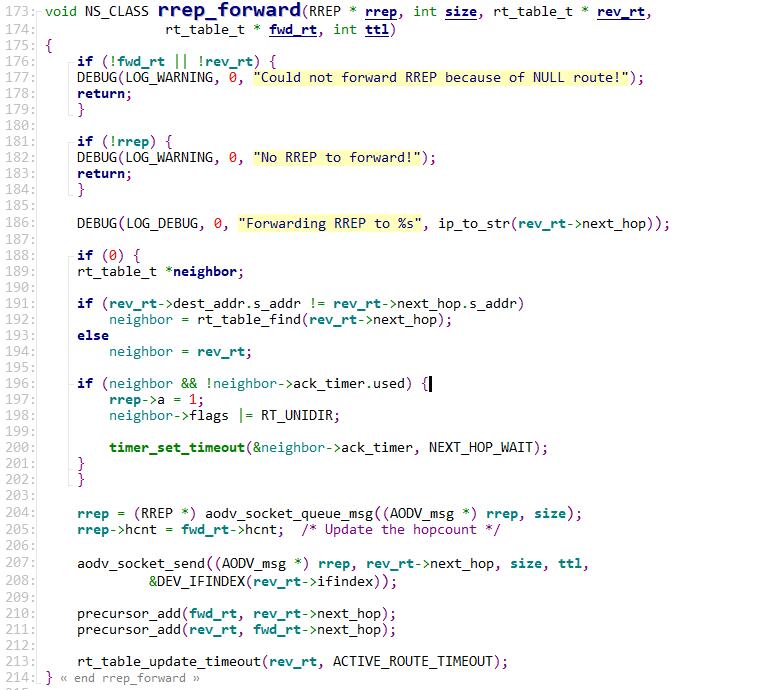
143-155：如果邻居存在并且邻居的状态是有效的，计时器还没使用，那么就将包的flag设置为RREP\_ACK，邻居标记为非单向，移除邻居的hello计时器，设置ack计时器。

157-159：更新路由表项

161-161：交由aodv\_socket\_send函数去发送消息

164-167：更新路由项fwd\_rt和rev\_rt的前驱节点

### rrep\_forward函数-转发rrep消息



176-180：当路由表项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

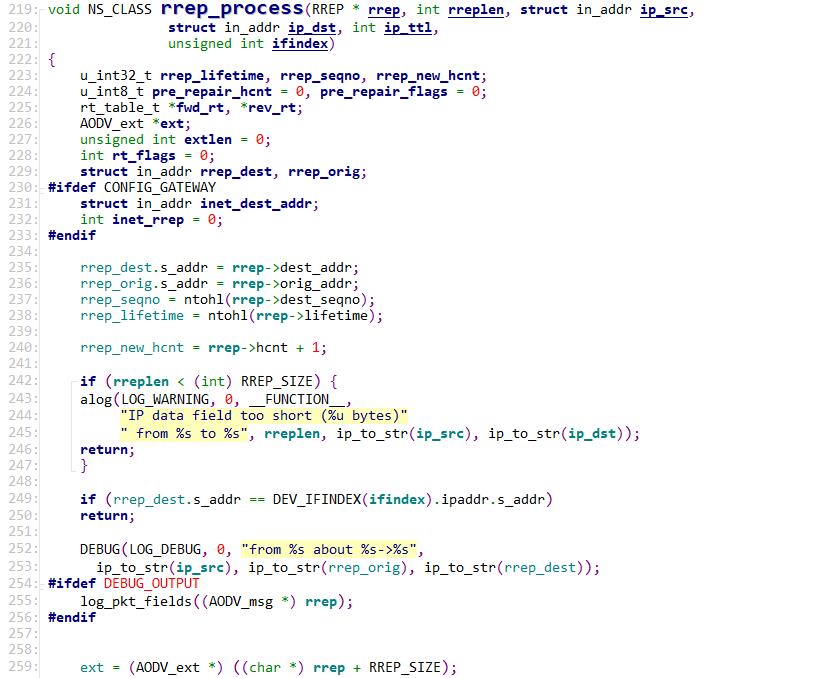
192-195：如果RREP消息的源IP地址不是相邻节点，那么我们必须找到通向源节点的下一跳IP地址。

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

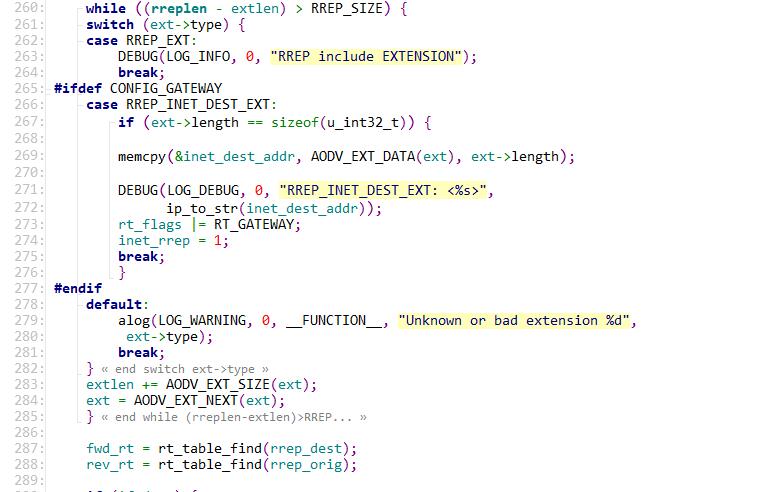
当套接字接收到RREP回复帧的时候，经过初步的过滤操作，最终会递交给这个函数进行处理。一个节点接受到RREP回复帧的时候需要判断是否是自己发出的RREQ回复的，如果不是则查询路由表项，将RREP重新单播回去。



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



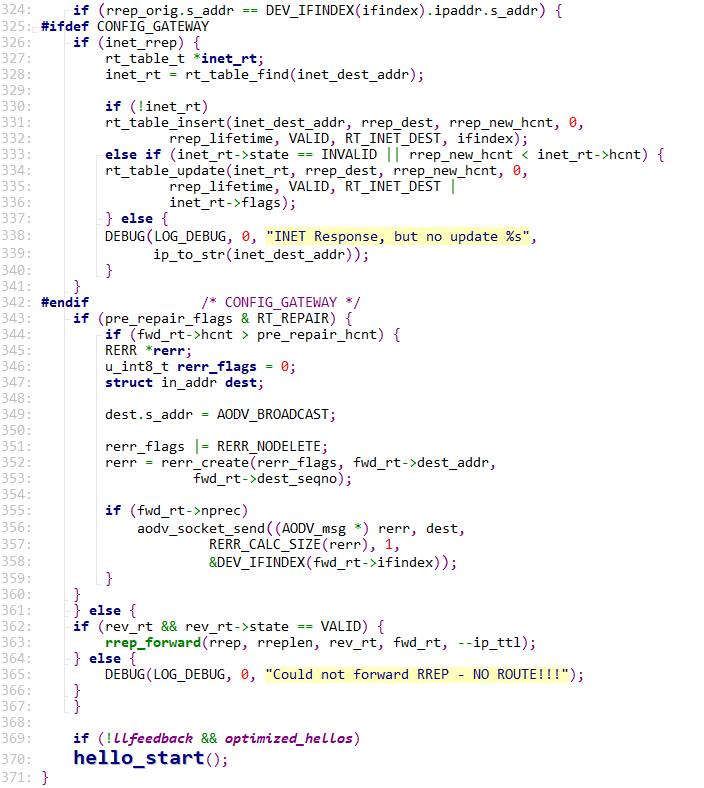
260-285：如果包的长度减去扩展长度仍大于RREP包的设定大小则进入该循环。该循环对RREP扩展和目的节点的拓展进行处理

287-288：获取有关RREP目的节点和源节点的路由表项以待后续使用



290-312：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

313-320：检测是否需要发送RREP\_ACK，需要则创建RREP\_ACK包，然后递交给发生函数发送



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

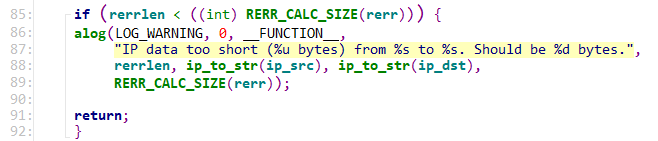
|  |  |
| --- | --- |
| 名称 | 功能说明 |
| RREP \*NS\_CLASS rrep\_create | 该函数通过传入的参数设置RREP包的prefix，hop count，目的IP和对应序列号，源IP和对应序列号等信息，返回一个新创建好的RREP包 |
| RREP\_ack \*NS\_CLASS rrep\_ack\_create() | 创建RREP\_ACK控制帧 |
| AODV\_ext \*NS\_CLASS rrep\_add\_ext(RREP \* rrep, int type, unsigned int offset, int len, 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

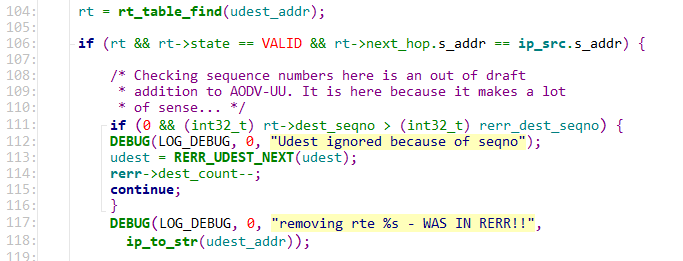
### rerr\_process函数-处理

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



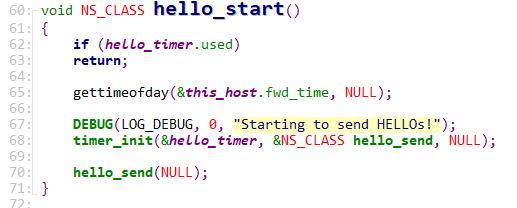
190-209：发送新生成的rerr消息。若只有一个目标节点，则单播该消息。否则，将rerr消息逐个发送到受影响的节点中。因为本段代码将ttl=1，所以仅发送给本节点的邻居节点。

### 其他函数

|  |  |
| --- | --- |
| void rerr\_add\_udest | 根据传入的参数向rerr消息中添加一条不可达节点 |
| RERR\_UDEST\_FIRST | 检查不可达的目的节点 |
| RERR\_UDEST\_NEXT | 查到一个不可达节点后，继续检查下一个不可达节点 |
| rerr\_create | 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。 |

## aodv\_hello.c

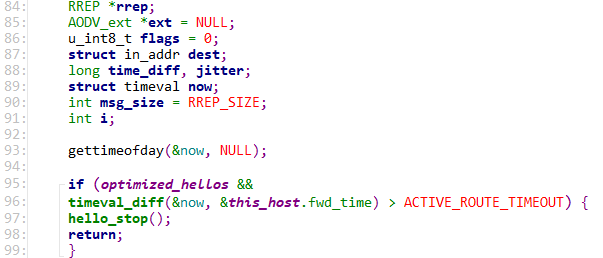
### hello\_start()函数-启动hello消息



62-63 若定时器处于开启状态，直接返回

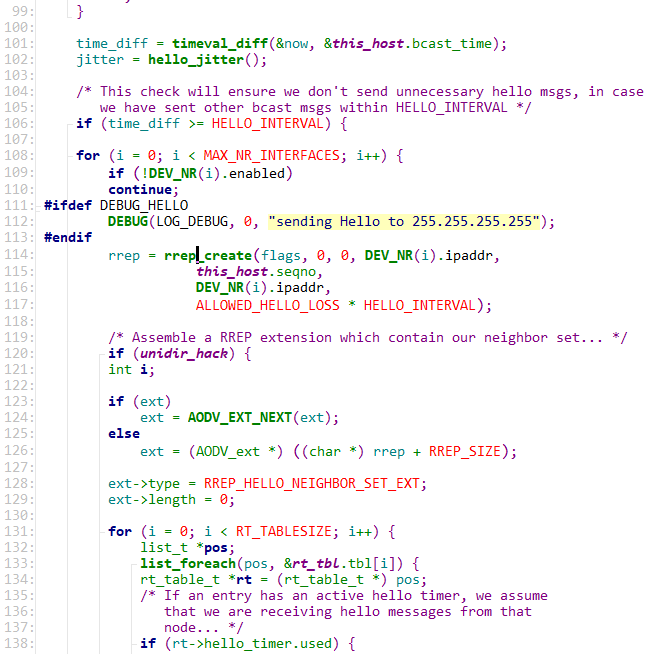
65-70 开启hello消息计时器，首先获取系统当前时间，设定DEBUG参数，表示开启发送hello消息的操作。68行，启动hello消息定时器，70行，将hello消息发送出去。

### 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。

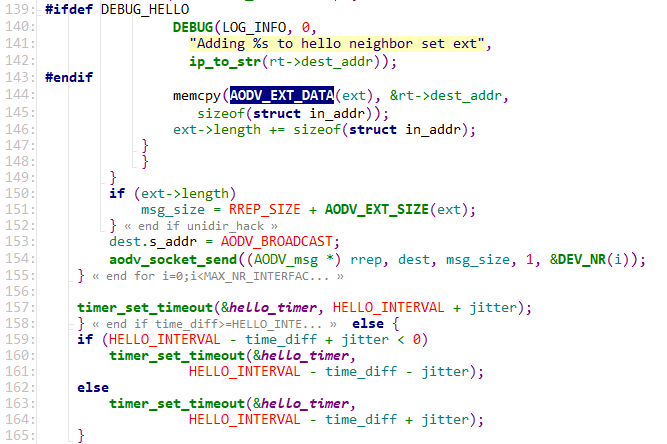
91-97：获取系统当前时间，存放于now中。计算当前时间与上次发送hello数据包的时间差，若大于路由超时时间，则说明节点和邻居节点的链接处于断开状态，调用hello\_stop函数，关闭hello消息，函数返回。



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。

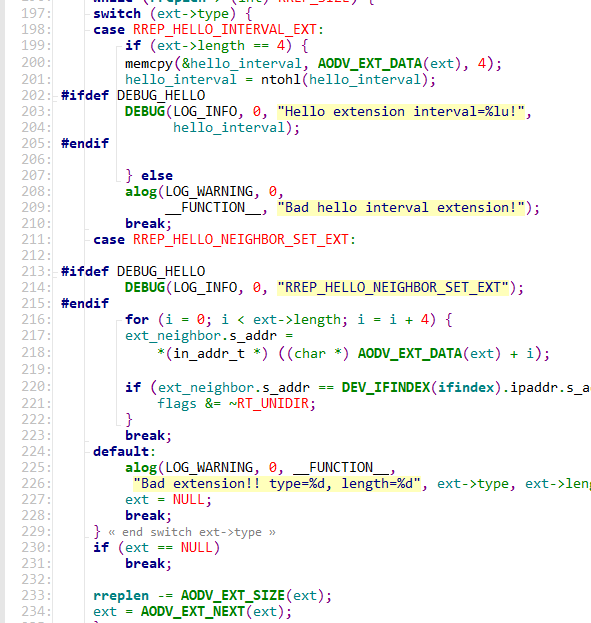
148-157：初始化数据包的大小，将组装好的rrep消息，广播给所有的邻居节点。调用time\_set\_timeout函数，设定计时器时间为HELLO\_INTERVAL + jitter；

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。

248-248：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。

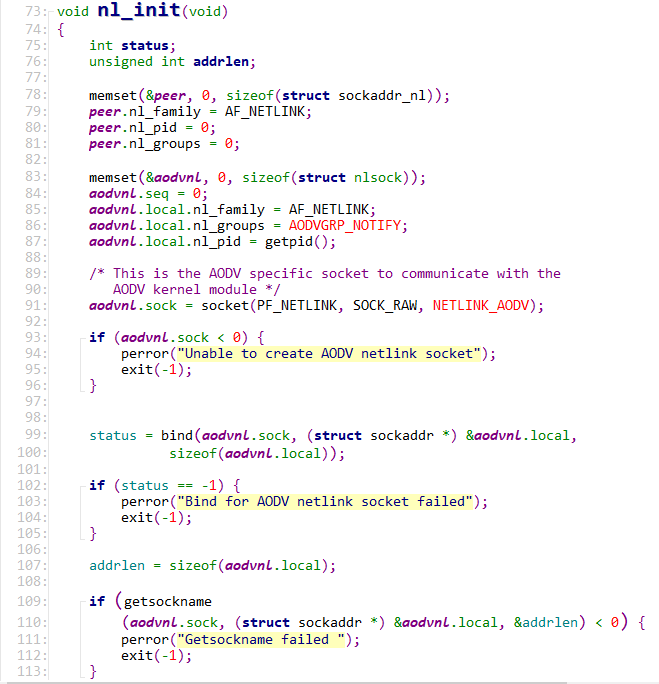
265-282：在路由表中找到对应的表项后，则通过执行hello\_update程序，增加该表项的生命周期。同时，若这条路由表项包含hello消息中的目的序列号，则表项中hello\_cnt+1，否则，通过调用rt\_table\_update()函数，将hello消息中的目的序列号添加到路由表项中。

### 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。

075-087： 将aodvnl，peer分别进行初始化

091-091： 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，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

093-096： 如果创建套接字失败，则输出信息，并返回-1

099-100： 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

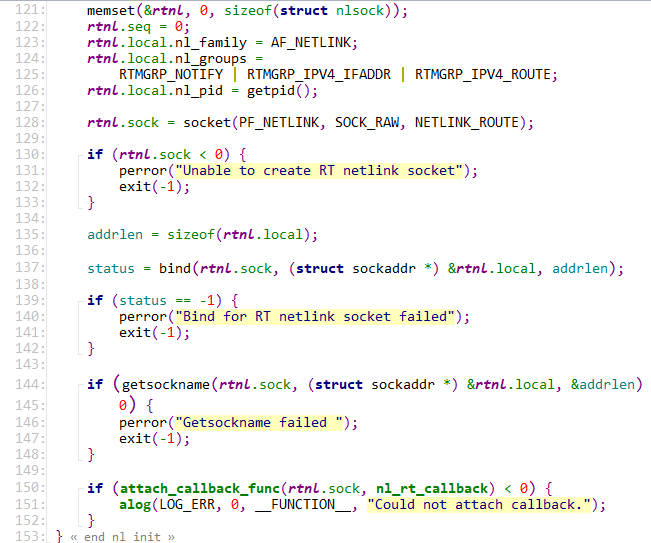
102-105： 如果bind函数出错则输出Bind for AODV netlink socket failed，并返回-1

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

128-128：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，则输出信息并返回-1退出

137-137：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，则输出信息并返回-1退出

144-148：如果获取rtnl.sock的本地地址失败，则输出信息并返回-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

本函数根据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

nlmerr：nlmsgerr类型的指针该结构体携带有错误消息

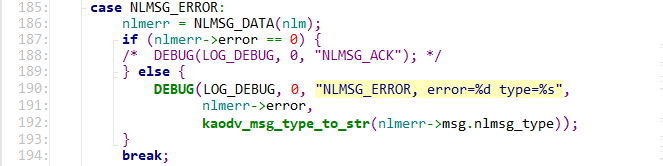
dest\_addr，src\_addr：in\_addr类型的指针，分别表示目的IP和源IP地址

m：kaodv\_rt\_msg\_t类型的指针，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

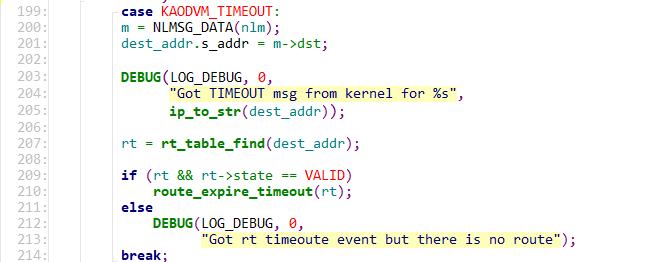
184-184：根据内核套接字nlm接受到的不同命令类型选择不同的处理方式



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

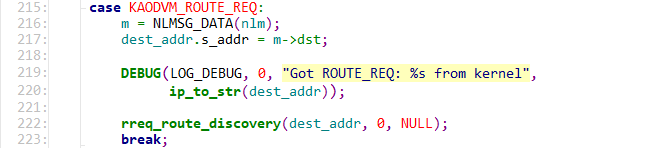


196-198：如果消息类型为KAODVM\_DEBUG则输出输出附带的调试信息

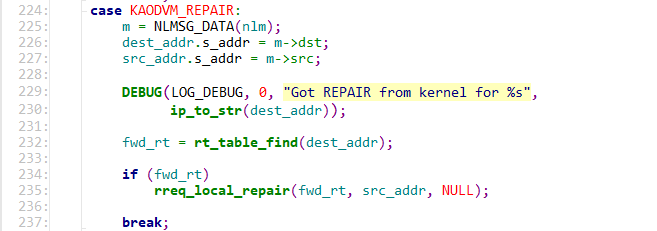


199-214: 如果消息类型为KAODVM\_TIMEOUT,如果路由表中目的地的路由表项有效，则将它

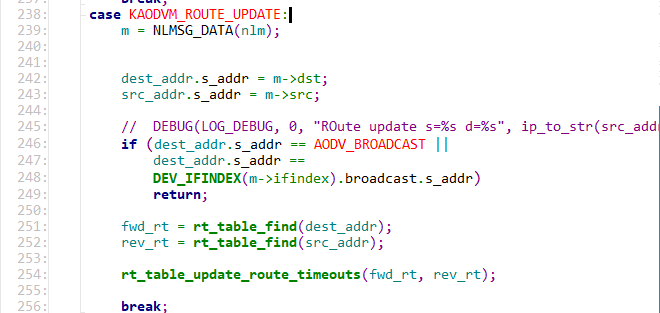
设置为超时无效状态



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

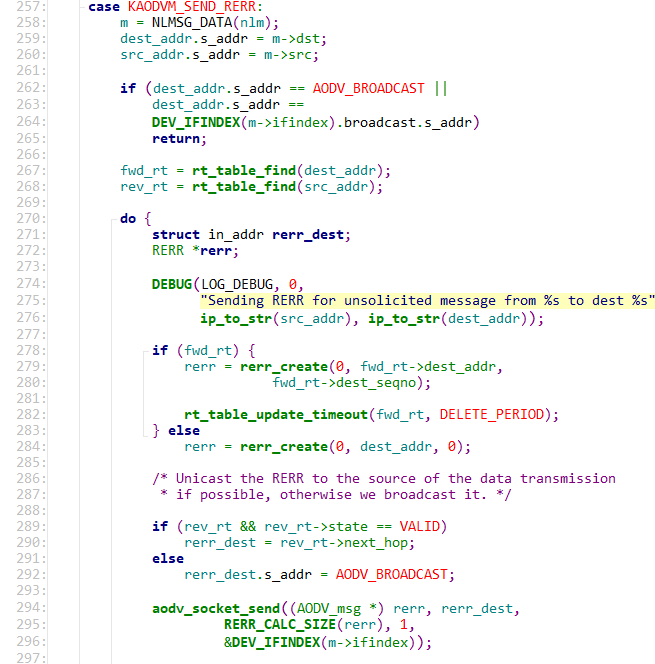


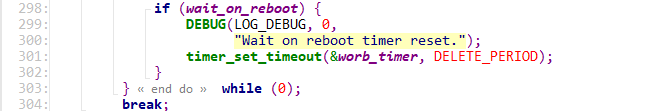
224-237: 如果类型为 KAODVM \_REPAIR且目的地址在路由表中，开始本地修复过程



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





257-257： 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

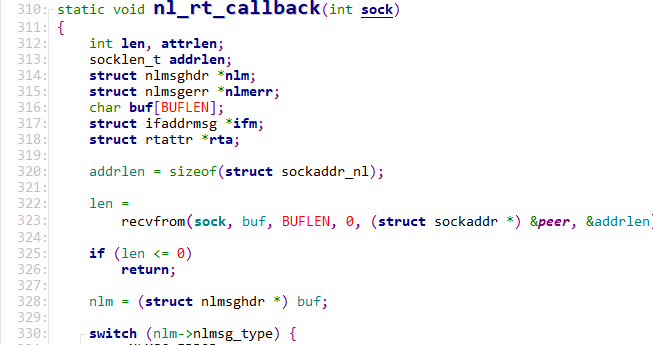
型，并且更新该路由表项的过期时间，否则便创建一个将序列号置为0的rerr类型的消息

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

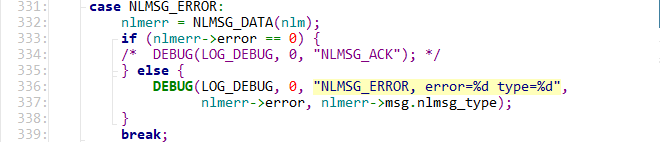
nlmerr：nlmsgerr类型的指针

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

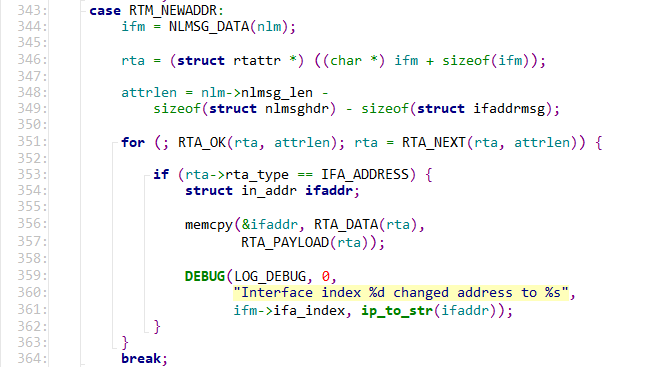
rta：rtattr类型的指针，创建或者删除一个特定的网络接口，或者从一个特定的网络接口上获得信息会使用到该类型

322-326： 如果recvfrom调用失败，则返回空

330-330： 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

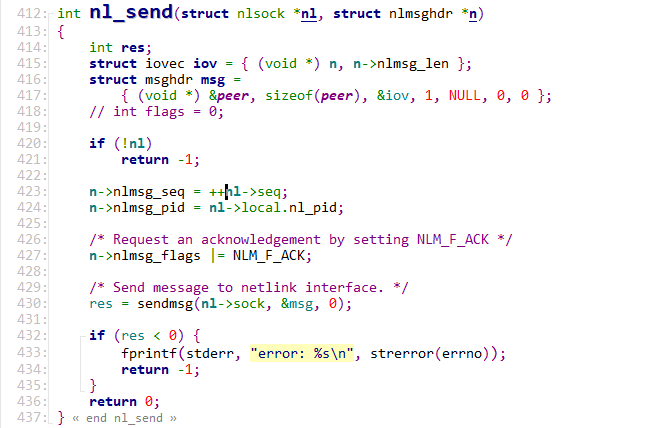


343-364: 如果值为RTM\_NEWADDR，则首先为ifm、rta、attrlen赋初值。在rta队列中寻

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

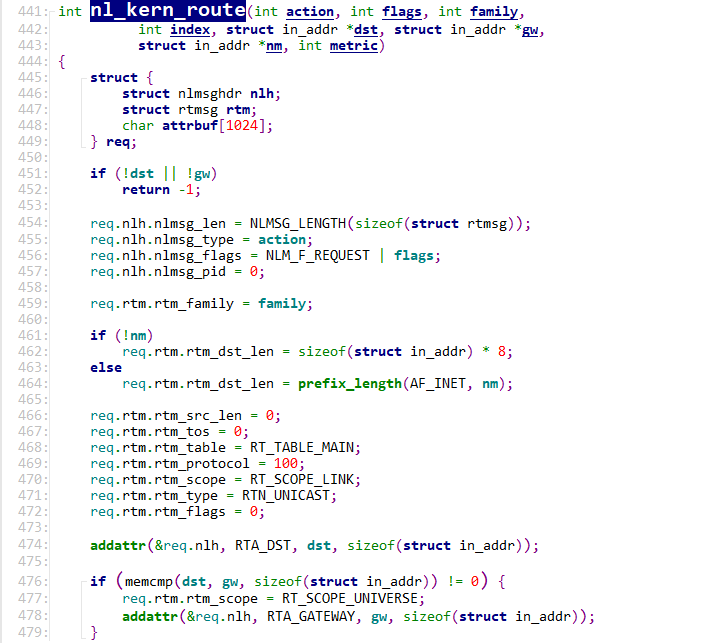
423-424：序列号加1，设置目标的pid

427-427：通过设置NLM\_F\_ACK来请求确认

430-430：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



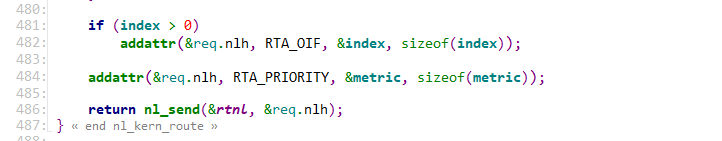
445-449：定义结构体req，包括一个nlmsghdr结构体,一个rtmsg结构体和一个char数组

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，并再添加一个类型

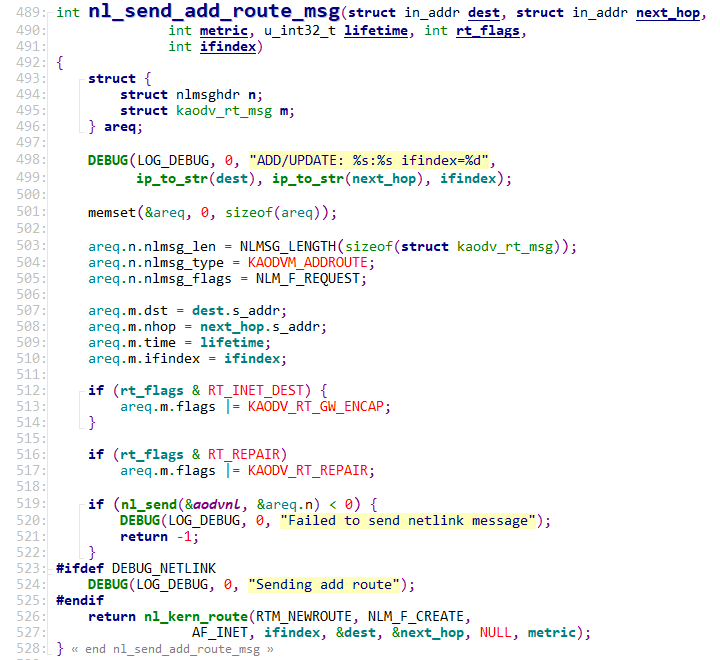
RTA\_GATEWAY的路由表项



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

484-484：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）

501-501：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

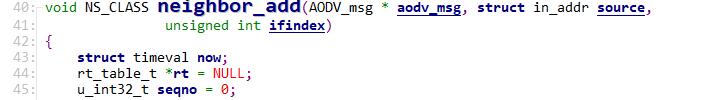
519-521：如果nl发送失败，则打印消息后退出函数返回-1

### 其他函数

|  |  |
| --- | --- |
| 函数名 | 说明 |
| void nl\_cleanup | 关闭两个套接字 |
| int prefix\_length | 计算前缀长度 |
| int nl\_send\_no\_route\_found\_msg | 向内核发特定目标地址没有找到路由消息 |
| nt nl\_send\_del\_route\_msg | 向内核发删除特定路由的消息 |
| int nl\_send\_conf\_msg | 向内核发送设置参数 |

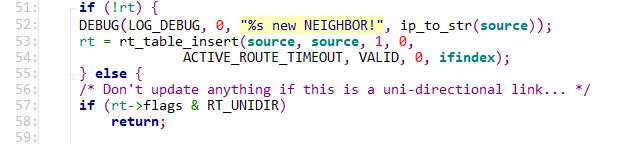
## Aodv\_neighbor.c

### neighbor\_add函数­——更新活跃的邻居节点（更新来自非HELLO AODV控制消息的邻居）



043-045：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：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。

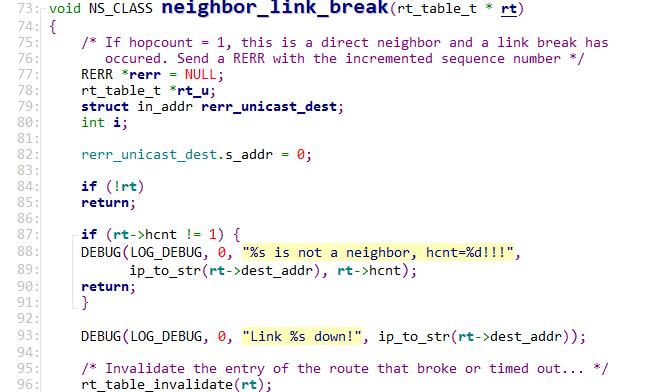
056-059：判断如果为单向链路直接返回，不用更新路由信息。

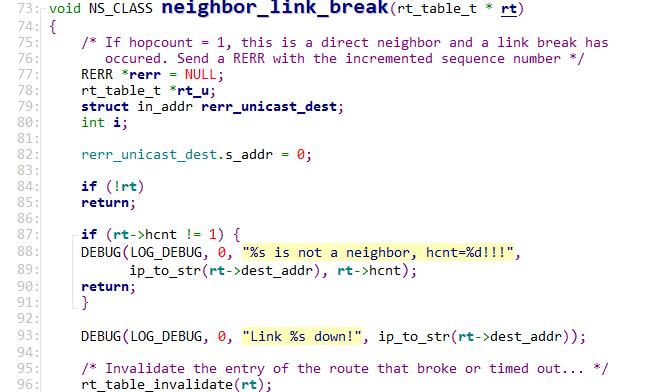


067-068：表示如果路由信息是存在且活跃的，则需要用ALLOWED\_HELLO\_LOSS \*

HELLO\_INTERVAL来添加对应路由的活跃周期。

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





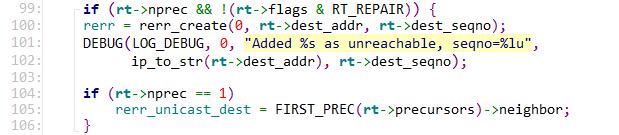
084-086 ：判断路由表是否为空若为空则直接返回。

087-092 ：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。

096-096 ：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。

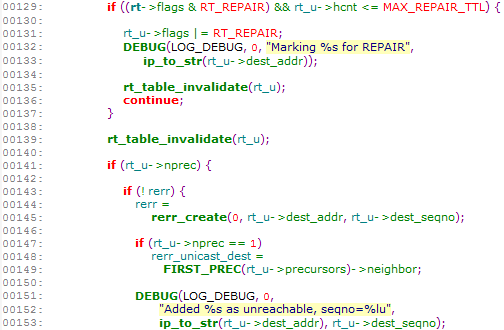


099-106 ：除去路由被回复的情况，创建一个路由错误信息rerr。调用rerr\_create()函数来

传入目的地址和序列号。



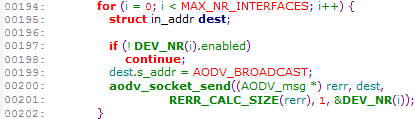
109-110 ：用来清除了路由前驱列表。

129-152：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：将损失的链接标记为要修复。

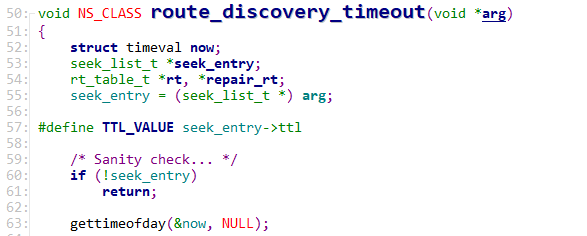
139-153：将损失的链接标记为要修复的情况下，在对其他所有不可达条目应执行相同的操

作。

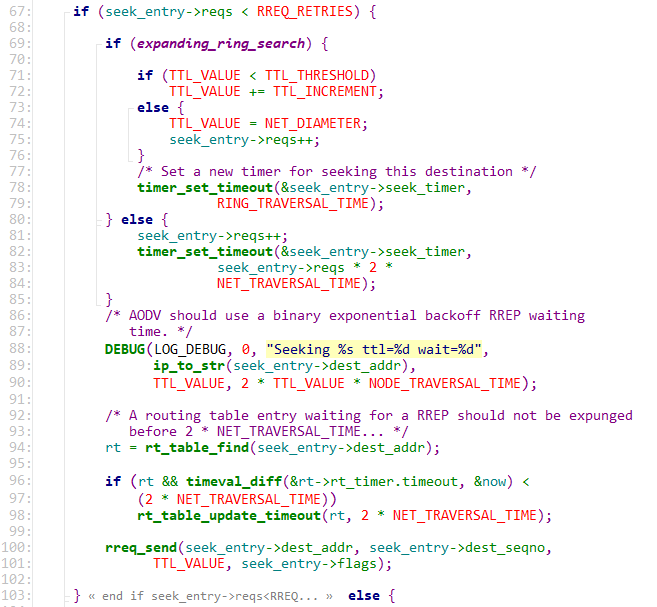
194-202： 表示只在不可达路由结点的先驱结点的接口上传输RERR。

## Aodv\_timeout.c

### route\_discovery\_timeout()函数-路由发现超时

52-63： 判断是否包含带查找的路由表项，若没有，系统返回；否则获取系统当前时间并

记录在now中。



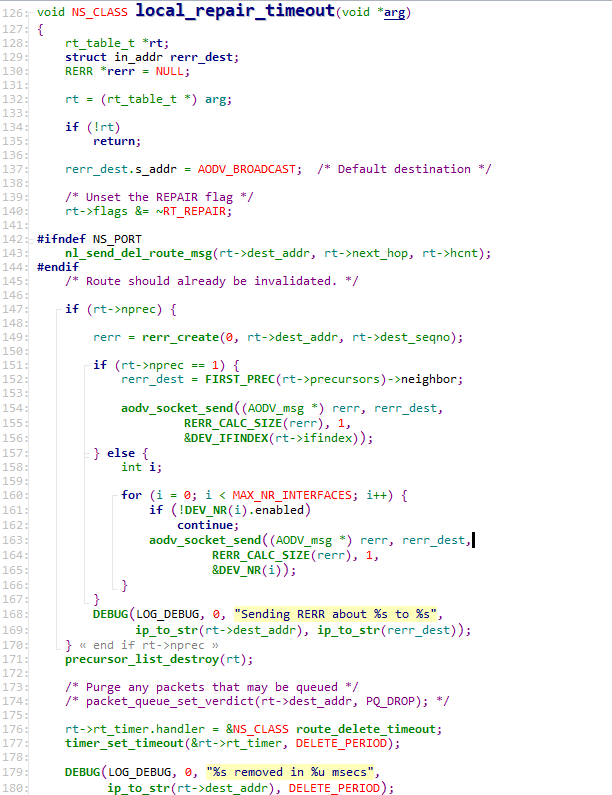
067-103：若尚未满足超时要求，则创建一个新的计时器，调用time\_set\_timeout函数，将

新计时器的时限设置为两个NET\_TRAVERSAL\_TIME；若已经超时，就直接返回没有发现路由

信息。

118-121：若所寻找的路由正处于修复状态，则直接返回超时信号。

### local\_repair\_timeout（）函数-本地修复超时



128-165：若路由表非空，则将目的地址设置为默认的广播地址，设置这条路由的标志位

为REPAIR，同时将这条路由消息无效的消息广播给网络中的其他节点。

171-180：如果修复超时，调用precursor\_list\_destory（）函数，清空队列中所有可能引用

修复路径的数据包 。

### hello\_timeout()函数-hello消息超时



227-258： 如果长时间没有收到hello消息，则调用该超时函数。如果能够对路径进行修复，

则设置路径的标志位，同时调用neighbor\_link\_break函数，将这条路径设置为断开状态。

### 其他函数

|  |  |
| --- | --- |
| void route\_expire\_timeout | 路由到期超时函数，路径生命周期结束，删除对应的路由表项 |
| void route\_delete\_timeout | 路由删除超时函数，从路由表中删除路由信息 |
| void rrep\_ack\_timeout | rrep确认超时函数，若ack传送失败，则将目的地址加入黑名单 |
| void wait\_on\_reboot\_timeout | 重启等待超时函数，若重启超时，只将此记录于日志文件。 |
| void packet\_queue\_timeout | 数据包排队超时函数，调用time\_set\_timeout函数，处理排队超时的数据包 |

# AODV仿真

## 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。

## 获取trace文件

### 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。

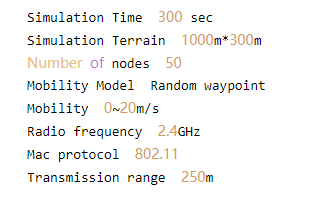
在终端~/ns-allinone-2.35/ ns-2.35/indep-utils/cmu-scen-gen/setdest/路径下输入命令：



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。

### 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：

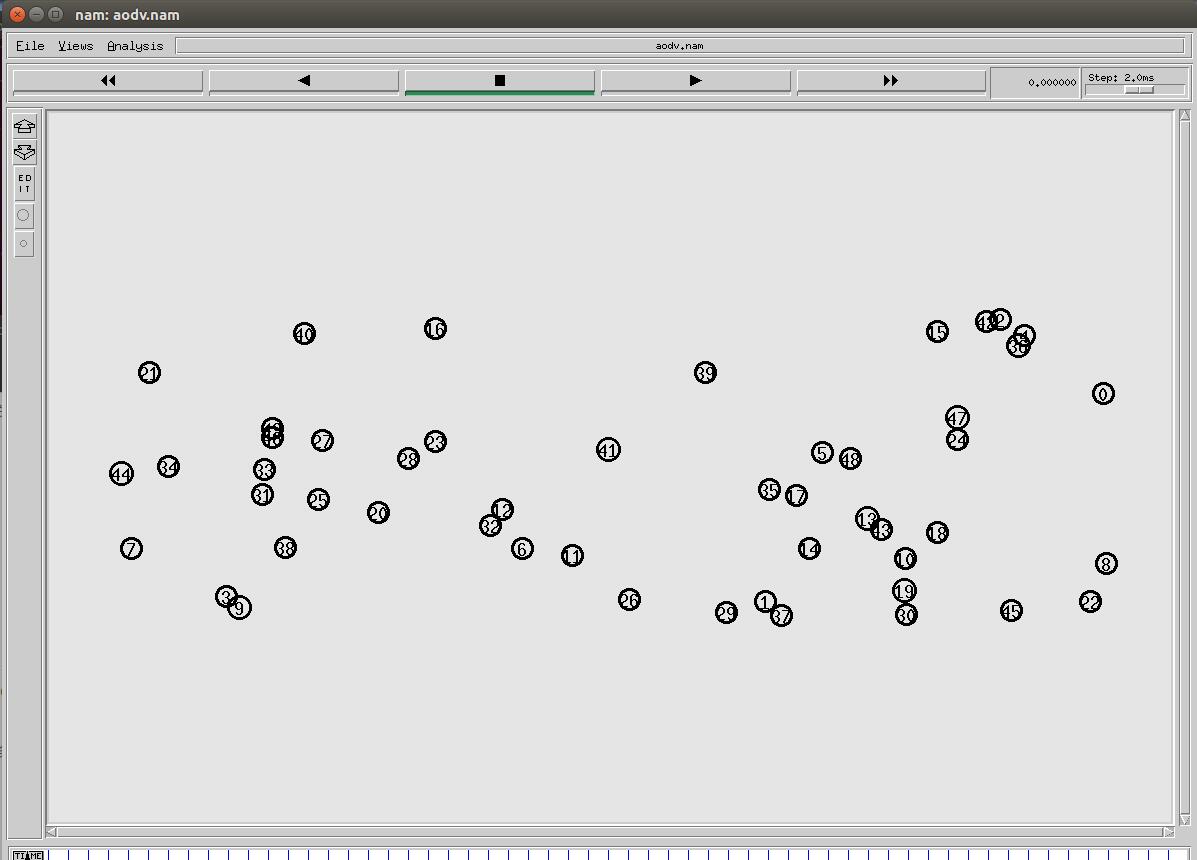


### 执行aodv.tcl脚本

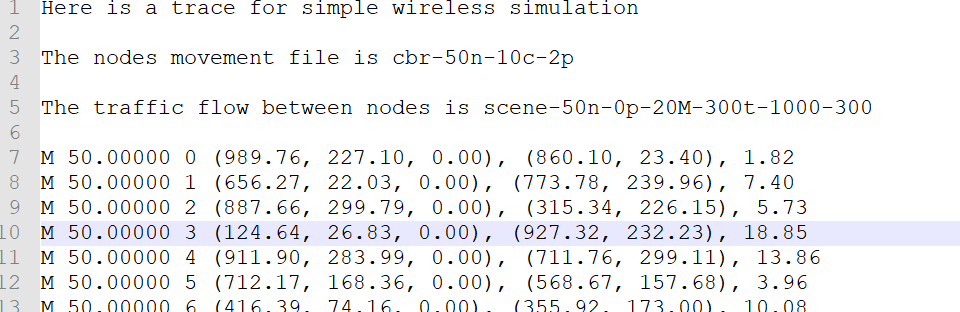
将以上生成的通信场景文件cbr-50n-10c-2p和移动场景文件scene-50n-0p-20M-300t

-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：



## 分析trace文件

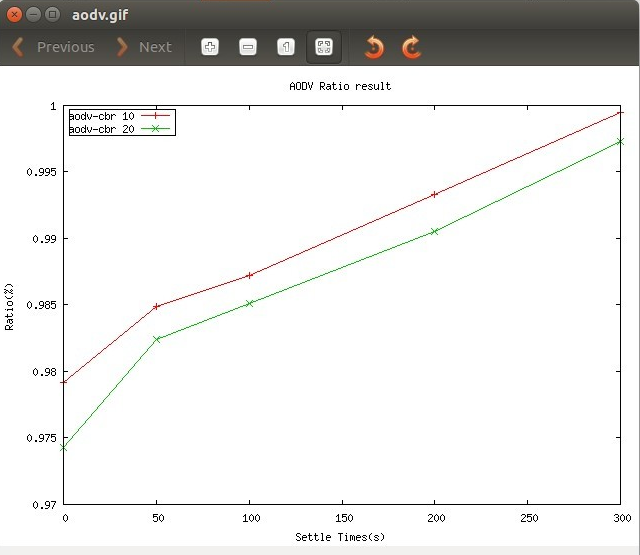
生成的trace文件一般都比较大，所以使用average.awk, frequency.awk，getRatio.awk, delay.awk, load.awk, run六个脚本程序对数据进行处理，获取分组投递率、路由发起频率、路由开销、平均时延四个表征协议性能的参量。

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：

### 分组投递率

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：

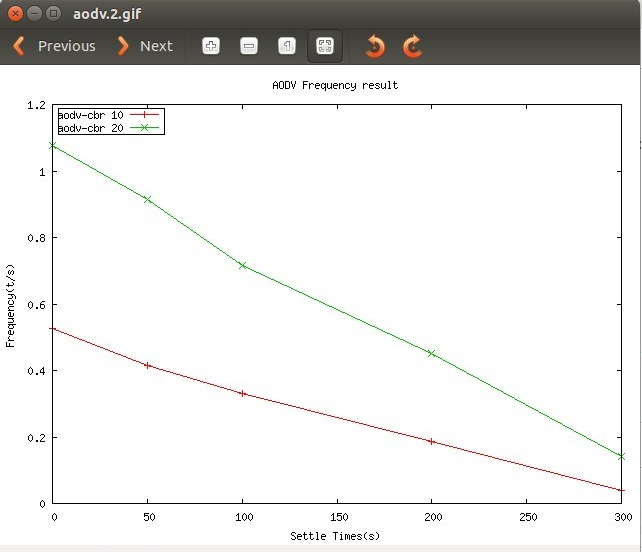
1. 随着节点停留时间逐渐增大，两种情况的分组投递率都增大，节点间路由出现断裂的情况减少。当停留时间为50s左右时，分组投递率的增长趋势发生明显变化，说明在这一过程中，节点与节点之间的路由信息变化较为剧烈；超过50s后，增长趋势趋于平稳，说明节点与节点之间的路由信息基本稳定，保持分组投递率的增长一直到仿真实验300秒全部结束。
2. CBR源较少的网络要比要比CBR源较大的网络分组投递率高。说明，减小CBR源的个数，可以提高数据包的分组投递率。

### 路由发起频率

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。

### 路由开销

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：

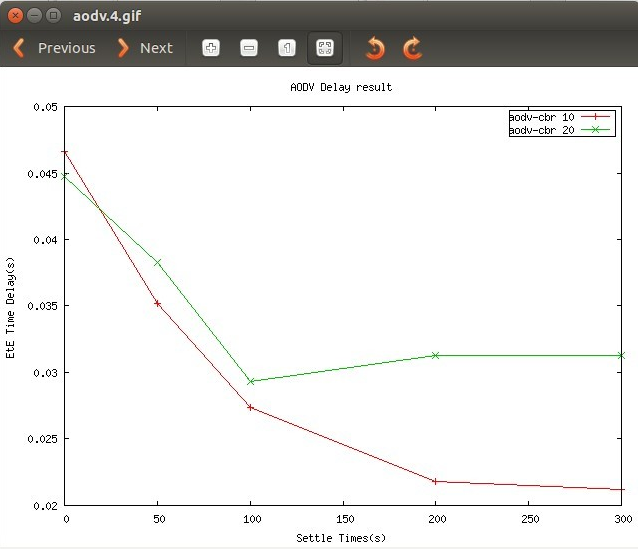
1. 随着节点停留时间的增大，归一化路由开销逐渐减小。说明当节点移动速度较慢时，在接收到的所有数据包中，用于路由发现和维护的分组减少，路由开销也减小。可以观察到，路由开销-节点停留时间曲线与路由发起频率-节点停留时间曲线的趋势大致是相同的，因此可以得出结论，路由发起频率和路由开销是正相关的。
2. CBR源多的网络，其路由开销也大。说明产生数据多的网络中，由于路由状态的不稳定，其用于发现和维护路由的分组较多。

### 平均时延

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。