网络协议栈分析

ADOV协议

软网1703 张兆轩 201792383

软网1703 高树达

任务分工详情：

张兆轩：

前期工作：通读RFC文档、查阅多篇国内外文献，分析总结以markdown的形式提交到Github。

中期工作：分析整个源代码的文件架构，功能模块，搞清楚代码运行的基本原理，开展文档的编写。

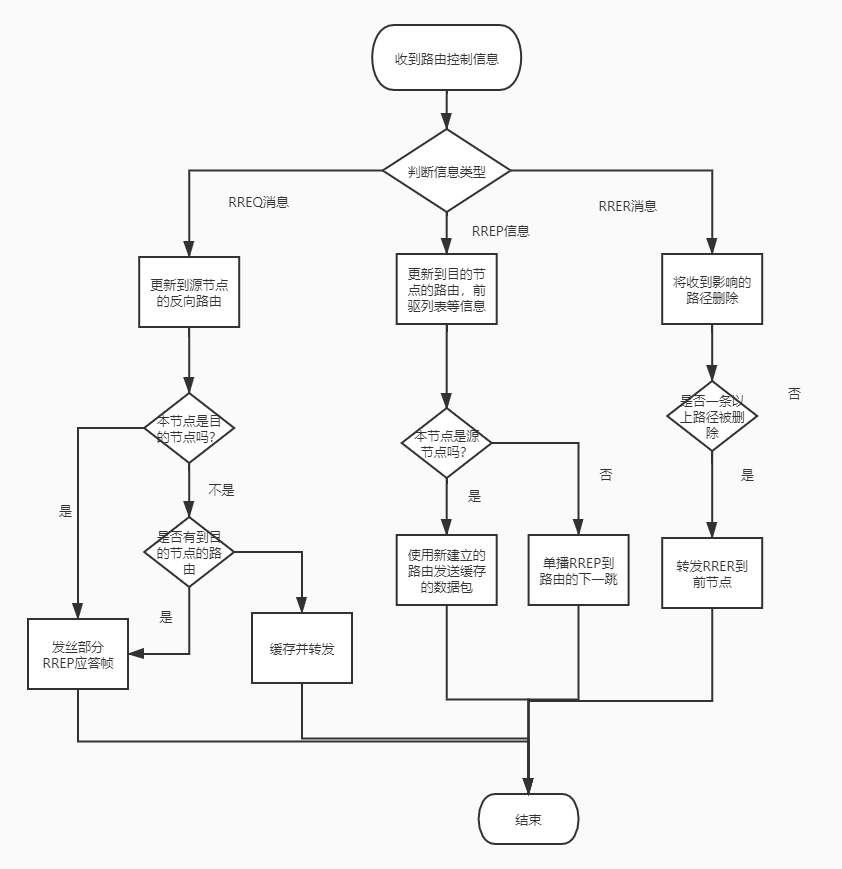
后期工作：完成大部分文档编写任务，整合另一名同学的分析结果，完成本次作业。

高树达：

# 一、概述

AODV算法只在多个网络节点中建立和维护一个动态的，自启动的，多跳路由的专属网络。ADOV使得移动检点能快速获得通向新的目的节点的路由，并且节点仅需要维护通向它相邻一跳节点的信息。它在路由表中引入目的节点序列号（dest\_seqno），确保了任何时候都不会出现回环。同时，引入HELLO报文机制，当网络中出现连接断开或其他异动时，AODV能够让网络检点实时对这些变化做出响应，避免了传统路由算法出现的很多问题。

AODV节点的工作流程如下图所示：



## 协议报文以及路由过程

|  |  |  |
| --- | --- | --- |
| 报文名 | 解释 | 用途 |
| REEP | 路由请求 | 当某结点接收到新的目标节点时广播REEQ消息 |
| RREP | 路由回复 | 当一个节点向给它发送RREQ的节点单播一条RREQ时，原节点将相邻节点状态信息存入路由表 |
| REER | 路由错误 | 若活动路由表某一条连接发生断裂，REER将指出不能到达的节点或子网 |

### RREQ帧

当源节点S需要向目的节点D发送数据包时，但有没有目的节点D的路由入口时，会发送RREQ帧，RREQ帧会广播。

1. 对REEQ的处理

接收到RREQ 的结点做如下处理：  
（1）创建一个表项，先不分配有效序列号，用于记录反向路径。  
（2）如果在“路由发现定时”内已收到一个具有相同标识的RREQ报文，则抛弃该报文，不做任何处理;否则，对该表项进行更新如下：  
I.下一跳结点=广播RREQ的邻居。  
II.跳数=RREQ报文的“跳计数”字段值。  
III.设置表项的“过时计时器”。  
（3）如果满足以下条件，则结点产生“路由回答报文”RREP，并发送到信源；否则更新RREQ报文并广播更新后的RREQ报文。  
I.该结点是信宿。  
II.结点的路由表中有到信宿的活动表项，且表项的信宿序列号大于RREQ中的信宿序列号。  
（4）更新RREQ报文并广播更新后的RREQ报文  
I.信宿序列号=本结点收到的信宿相关的最大序列号。  
II.跳计数加1。

1.帧的格式

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Type |J|R|G|D|U| Reserved | Hop Count |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| RREQ ID |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Destination IP Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Destination Sequence Number |

+-+-+-++-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+--+-+-+-+-+-+-+-+-+-+-+-+-+-+-++-+-+-+-+-+

| Originator IP Address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Originator Sequence Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

2.帧中字段的含义

type : 1 (默认1)

J : join

R : repair

G : Gratuitous RREP (免费路由回复标记) 说明是否应该向目标节点发送一个路由回复的报文

D : 仅允许目的节点进行回复

U : 指示目标节点序列号未知

Reserved : 填充0 会被接收端忽视的字段

Hop Count : 从发起节点到处理该节点请求的跳数

RREQ ID : ID

Destination IP Address : 目的节点IP地址

Destination Sequence Number : 目标节点序列号 发起节点在以前能通向目的节点的节点中找到的最新的序列号

Originator IP Address : 发起本条路由请求信息的IP地址

Originator Sequence Number : 发起者的路由表中现在正在使用的序列号

### 1.1.3 RREP帧

当RREQ到达时，目的节点发送的反向路由帧，使用这个帧来使网络上的各个节点建立前一个节点的路由，节点只对收到的第一次RREQ产生反应，重复发送的RREQ将不能产生重复的RREP帧。

接收到RREP的操作：

I.如果收到相应的RREQ的信宿序列号与信宿维护的当前序列号相等，则信宿将自己维护的序列号加1，否则不变。

II.跳计数=0。

III.定时器值。

（2）中间结点产生的RREP

执行如下操作：

I.本结点获取的该信宿的最大序列号。

II.跳计数=本结点到信宿的跳数（查相应表项即可得到）。

III.更新本结点维护的“前向路由表项”的下一跳和“反向路由表项”的前一跳

b. 对RREP的处理

结点对接收到的RREP 作如下处理。

（1）如果没有与RREP报文中的信宿相匹配的表项，则先创建一个“前向路表”空表项。

（2）否则，满足如下条件对已有表项进行更新。

1. 帧的格式

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Type |R|A| Reserved |Prefix Sz| Hop Count |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Destination IP address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Destination Sequence Number |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Originator IP address |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Lifetime |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

1. 帧中字段的含义

type : 2 (默认2)

R : repair

A : acknowledgment required (需要确认)

Reserved : 填充0 会被接收端忽视的字段

Prefix SZ : 前缀长度 值为 0 - 31 前缀指的是 IPV4地址 - 主机地址的长度

Hop Count : 从发起节点到处理该节点请求的跳数

Destination IP Address : 目的节点IP地址

Destination Sequence Number : 目标节点序列号 从路由表中查找对应的序列号

Originator IP Address : 发起本条路由请求信息的IP地址

lifetime 生命时长 在规定的这段时间中 接受到这条消息的节点将认为这条路径是可用的

### 1.1.4 RRER帧

路由错误帧，用来进行路由的错误控制

执行操作：

邻居间周期性的互相广播“Hello”报文，用来保持联系，若在一段时间内没有收到“Hello”报文，则认定为链路断。例如当结点X、Y之间链路产生断路使数据无法通过此条链路传至信宿，则结点X会产生RRER报文向信源报告此情况。RRER通过广播形式传送，维护路由表的结点收到此报文会更新路由表（将X、Y间的路由设成无效），并转发RRER 报文。

1. 帧的格式

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Type |N| Reserved | DestCount |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Unreachable Destination IP Address (1) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Unreachable Destination Sequence Number (1) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

| Additional Unreachable Destination IP Addresses (if needed) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

|Additional Unreachable Destination Sequence Numbers (if needed) |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-

1. 帧中字段的含义

type : 3 (默认3)

N : No delete 当上一个节点对这个连接做了本地修复，就将这个标志位置真，也即不用删除这个路由

Reserved : 填充0 会被接收端忽视的字段

DestCount : 本消息内包含的不可达目的节点的数目

Unreachable Destination IP Address : 不能到的节点目的节点IP地址

Unreachable Destination Sequence Number : 与上面的目标节点对应的序列号

Additional : 可以包含多个不可达节点，用来表示多个不可达节点有一个 加一对儿消息

### 1.1.5 REEP ACK帧

收到需要确认的帧时用来回复的帧

1. 帧的格式

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

| Type | Reserved |

+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

1. 帧中字段的含义

type : 4(默认4)

Reserved : 填充0 会被接收端忽视的字段。

## 路由表

#### 路由表结构分析

struct rt\_table {

list\_t l;

struct in\_addr dest\_addr; /\* IP address of the destination \*/

u\_int32\_t dest\_seqno;

unsigned int ifindex; /\* Network interface index... \*/

struct in\_addr next\_hop; /\* IP address of the next hop to the dest \*/

u\_int8\_t hcnt; /\* Distance (in hops) to the destination \*/

u\_int16\_t flags; /\* Routing flags \*/

u\_int8\_t state; /\* The state of this entry \*/

struct timer rt\_timer; /\* The timer associated with this entry \*/

struct timer ack\_timer; /\* RREP\_ack timer for this destination \*/

struct timer hello\_timer;

struct timeval last\_hello\_time;

u\_int8\_t hello\_cnt;

hash\_value hash;

int nprec; /\* Number of precursors \*/

list\_t precursors; /\* List of neighbors using the route \*/

};

|  |  |  |
| --- | --- | --- |
| 变量名称 | 数据类型 | 作用 |
| dest\_addr(目的节点地址) | in\_addr | 用于标志使用此路由的最终目的节点，决定了数据分组转发方向。 |
| dest\_seqno(目的节点序列号) | u\_int32\_t | 反映此路由的新鲜度，一般序列号越大路由越新鲜， 这是保证开环的重要措施，在路由发现和路由应答更新路由时需要进行序列号的比较。 |
| flags(路由状态标志) | u\_int16\_t | 反映此路由目前的状态，主要用于告知数据分组经过此节点的时候处理方式。如果路 由处于无效状态，那么数据分组将丢失；如果处于修复状态，那么数据分组进入等待路由队列中；如果有效状态，那么直接转发。 |
| next\_hop(下一跳的ip地址) | in\_addr | 据分组经过本节点之后，数据分组将被直接转发的中继节点，通常下一跳节点应该出现在当前节点的邻节点列表中。 |
| hcnt(到达目的ip的跳数) | u\_int8\_t | 到达目的ip所需经历的子网的数量 |
| precursors(前驱节点表) | list\_t | 使用这条路由的所有直接前驱节点列表。在出现断链的时候可以通过前驱节点列表中是否存有节点而决定是否广播RERR消息 |
| rt\_timer(路由生命周期) | timer | 路由有效的生命期，在数据分组转发使用当前路由时会更新路由的有效生命期，当较长时间不使用此路由时，此路由的有效期将会过期，在路由管理时将会使路由失效 |

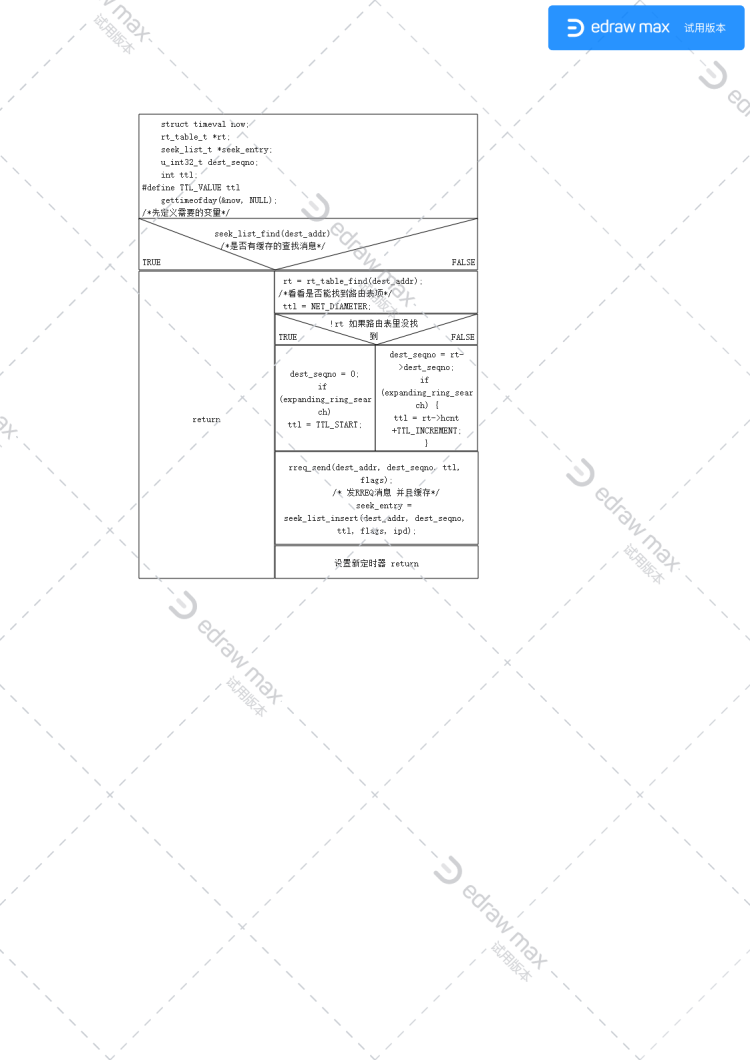
## 路由发现

当一个节点无法找到一个可用的到某个节点的路由时，它就会广播一条RREQ消息。中间节点各自更新到源节点的路由表，并维护指向发起节点的反向路由。目的节点和有效路由的中间节点产生RREP路由应答帧，该帧通过反向路由发送至源节点，源节点收到该帧后开始发送数据包，该过程称之为路由发现。

每当中间的节点发生移动时，将会导致其上游节点路由不可用，并发出RERR包通知源节点，从而使源节点重新开始路由发现过程。

与此同时，aodv协议也会从源节点定时广播hello报文来维护路由，发现某个链路不可达时，则会进行修复操作或者删除相应的记录。具体的修复方法为：假设节点一通过节点2、节点3发送数据包至节点4，当节点1把数据包交给节点2后，节点3突然消失了，这时节点2就无法侦测到节点3，断链产生。于是，节点2局部性的广播一个小范围RREQ去寻找节点3，并限制了到达节点3所经过的节点数。如果节点2通过RREQ仍然找不到节点3，则采用其他路由来到达目的节点。

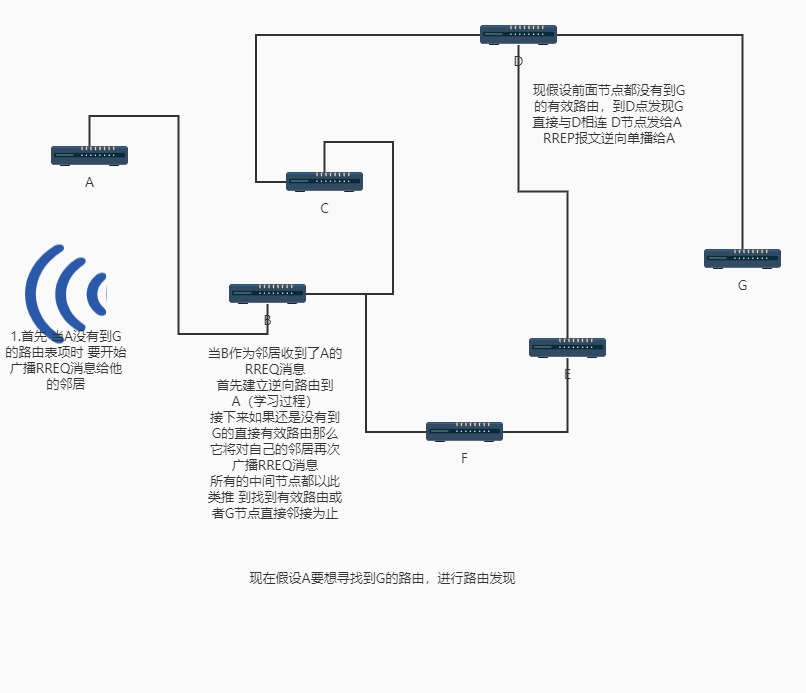
如果一条链路上一段时间一直没有数据包的发送，那么会认为这个路由已经过期，并将其删除。



## 1.2.3路由算法具体实现

AODV算法类似于计算机网络中的距离向量算法，当源节点有数据包发送时，先检查路由表中是否有直接到达该目的节点的路径可用，若有则直接发送该数据包，若没有，则将本节点信息封装至RREQ帧中并向网络中广播——(255.255.255.255)。

中间节点检查表中记录到REEQ帧中目的节点的路由是否有效，并且检查该REEQ帧的序列号是否小于路由表项中的RREQ，如果是的话返回一条路由应答帧（RREP），如果不是更新该表项序列号为RREQ中的序列号，转发该帧。

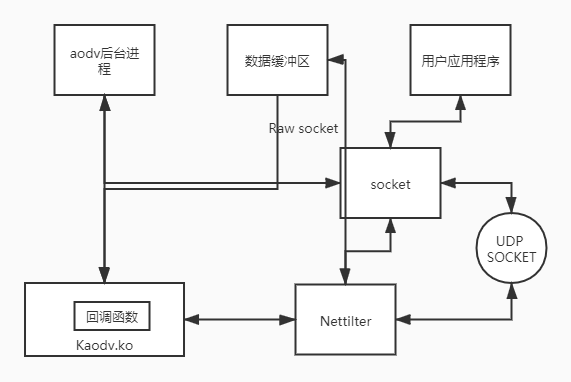


# 、二、代码分析

该源代码共有46个子文件，其中主文件夹包含aodv\_hello.c、aodv\_ne- ighbor.c等17个C文件，以及aodv\_rrer.h、aodv\_rreq.h等17个头文件，用于声明和定义aodv协议的工作流程。同时，在main函数中，声明了该协议栈的全局变量和全局函数。还包含了Makefile文件，用于编译源代码，README，用于简述代码使用方式和工作流程,。RFC（Request For Comments）3561，为该协议的说明文档。TAGS，用于编辑索引。子文件夹lnx中包含了AODV协议用于Linux的实现代码，前缀kaodv全称为kernel aodv即该实现应用于Linux内核。子文件夹patches包含了AODV的补丁文件，本次协议栈分析主要针对于主文件夹的C文件和H文件，子文件夹的内容不过多赘述。

## 2.1 文件介绍

该实例代码主要针对用户层，可把该过程简化为一个无线寻花读取事件队列的进程，路由的算法通过该进程实现，负责与其他节点进行消息的交互，建立和维护路由表。同时创建UDP socket套接字发送和接收，由aodv\_socket模块维护。各类控制信息的生成、发送、接收、转发和处理是aodv\_hello、aodv\_rerr、aodv\_rrep、aodv\_rreq等模块协作完成的。



## 2.1.1 用户层

|  |  |
| --- | --- |
| 文件名 | 简介 |
| list.h | 定义链表结构 |
| routing\_table.h | 路由表定义 |
| aodv\_hello.c | hello 消息的定义和相关操作 |
| aodv\_neighbor.c | 添加邻居节点并且处理邻居节点断开事件 |
| aodv\_rerr.c | rerr 消息的定义发送及处理 |
| aodv\_rrep.c | rrep 消息的定义发送及处理 |
| aodv\_rreq.c | rreq 消息的定义发送及处理 |
| aodv\_socket.c | 处理aodv的socket套接字 |
| aodv\_timeout.c | 超时处理 |
| list.c | 节点记录链表 |
| llf.c | 链路层分析 |
| main.c | AODV的全局变量定义以及运行源头 |
| nl.c | 完成用户层与内核层的通信 |
| params | 定义aodv草案里的参数 |
| endian.c | 选择网络字节顺序 |
| seek\_list | 记录RREQ索要发现到目的节点路由的IP地址 |
| locality.c | 查找主机模块 |

#### 2.1.2 核心层

|  |  |
| --- | --- |
| 模块名称 | 模块功能 |
| 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 | 显示核心词线程处理流程信息 |
| 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 |

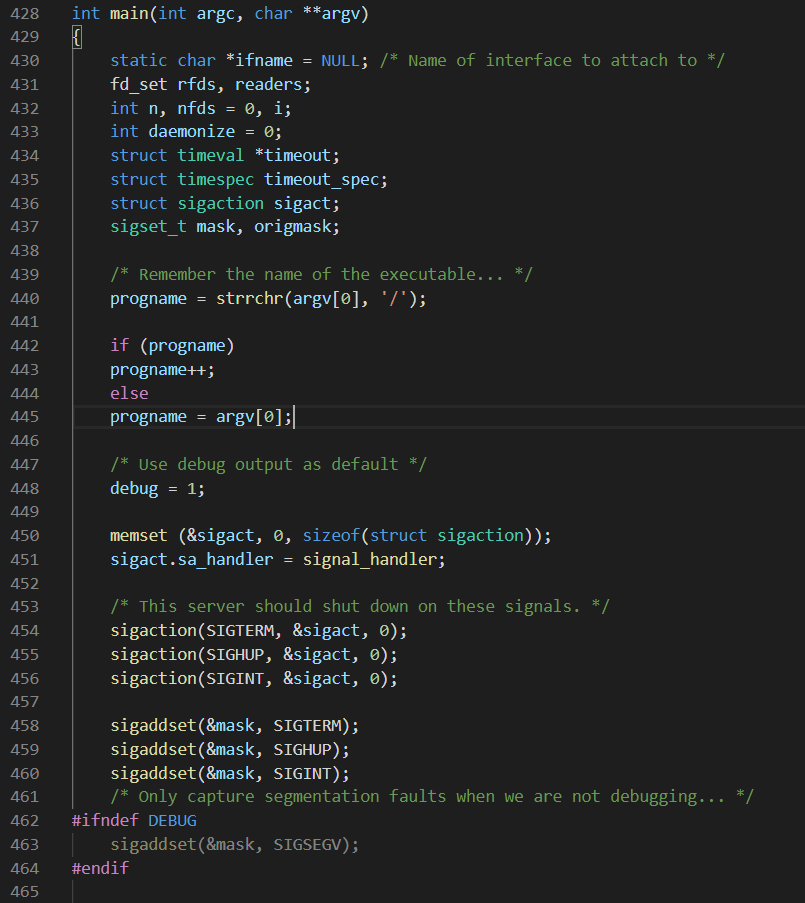
#### 全局变量（树达写）

|  |  |  |
| --- | --- | --- |
| 变量名称 | 数据类型 | 作用 |
| rreq\_gratuitous | int |  |
| ratelimit | timer |  |
| rt\_log\_interval | int |  |
| receive\_n\_hellos | int | 记录收到的hello报文数目 |
| optimized\_hellos | int |  |
| wait\_on\_reboot | int |  |
| qual\_threshold | int |  |
| ttl\_start | int |  |
| delete\_period | int |  |
| gw\_prefix | int |  |
| log\_to\_file | int |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## main.c

main.c可以简单的分成三部分，while循环之前的while循环之后的和while循环内部。主要功能为，初始化和与用户交互的作用。

## 2.3.1 while循环前部

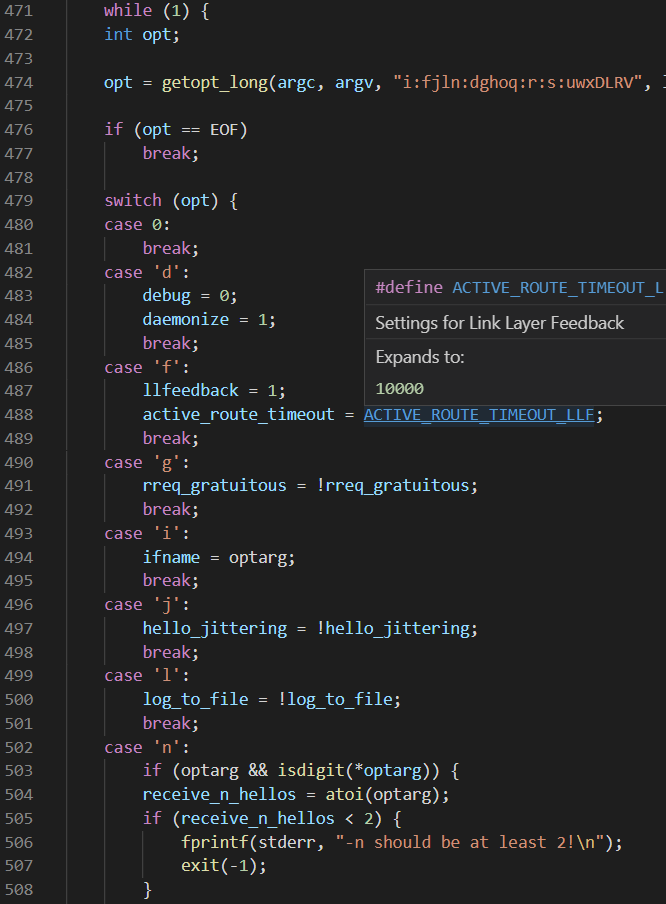


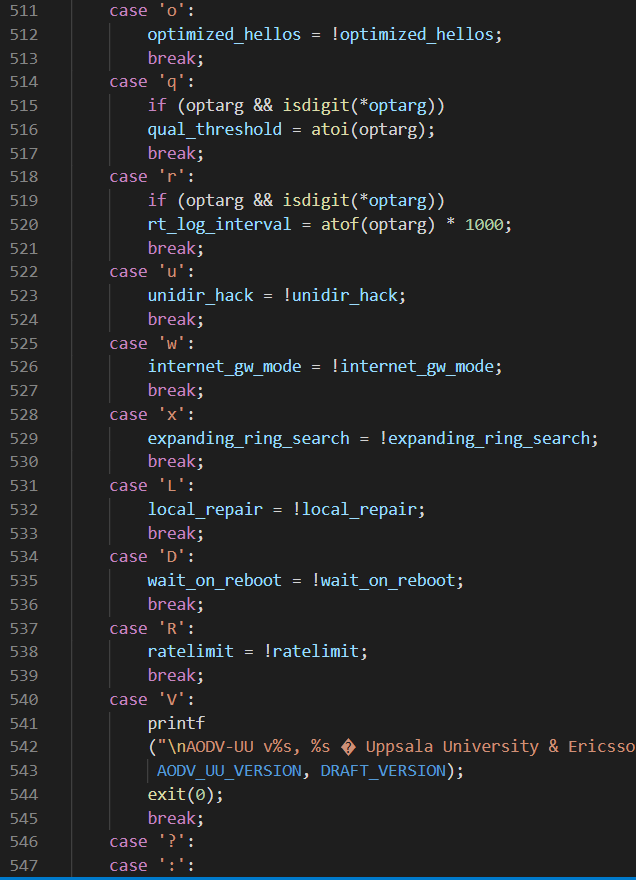
第一部分while循环之前的

430-437：定义一些全局变量

44-460：变量初始化

## 2.3.2 while循环呢内部



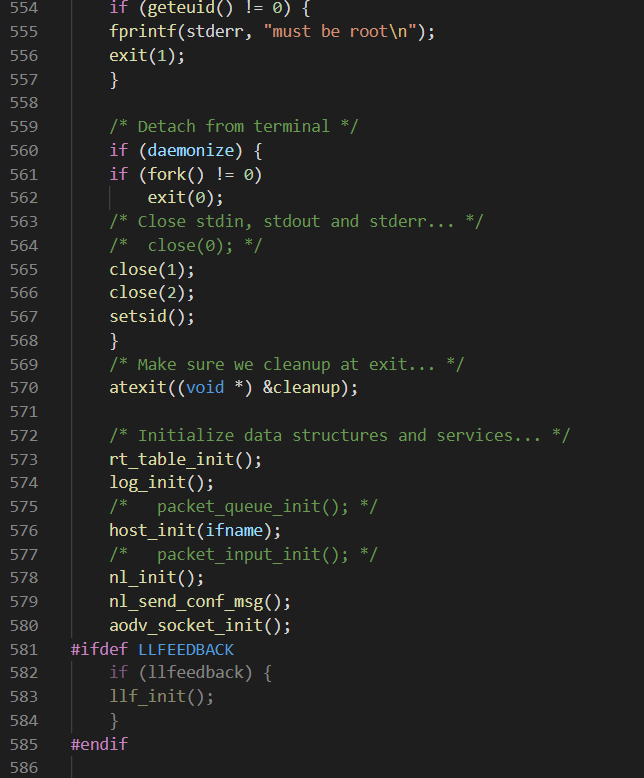


第二部分：while循环内部

该循环体作用主要是根据用户输入的不同参数执行固定的操作，下表是对应关系：

|  |  |
| --- | --- |
| 参数名称 | 功能 |
| -d | 守护进程模式 |
| -g | 在每个RREQ消息上强制设置gratuitous标志 |
| -h | 打印所有参数及含义 |
| -i | 表示要绑定的接口 |
| -j | 触发hello-jitter功能，默认开启 |
| -o | opt-hellos设置只在转发数据包的时候发送HELLO消息 |
| -l | log输出日志 |
| -r | log-rt-table每隔一段时间记录路由表 |
| -n | 接到n个HELLO消息之后才把该节点当做邻居 |
| -u | 侦测并避免单向链路 |
| -w | 开启实验性的因特网网关支持 |
| -x | 禁用RREQ消息的扩展环搜索法 |
| -D | 启用重启延迟的等待 |
| -L | 开启本地修复 |
| -f | 开启链路层反馈 |
| -R | 开启RREQ和RRER消息的速率限制 |
| -q | 为控制包设置一个信号质量的最小阈值 |
| -V | 输出版本信息 |

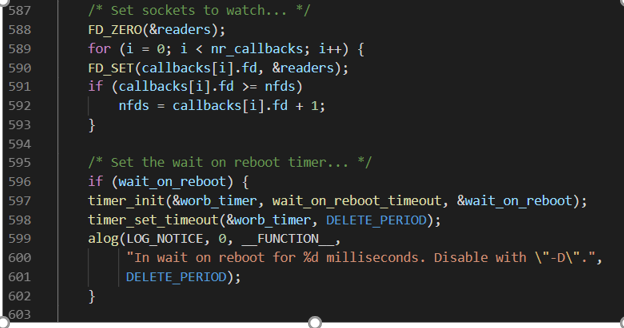
## 2.2.3 while循环后部



554-557：检测是否拥有超级用户权限

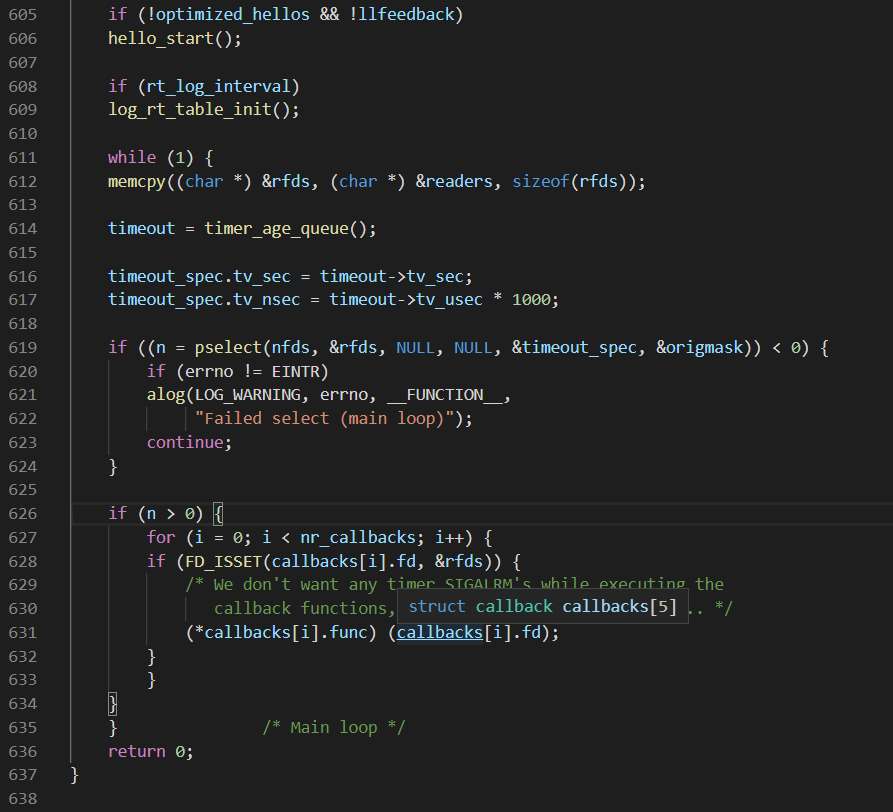
560-568：检查是否为-d启动

574-580：路由表、主机、套接字等资源初始化



588-590：清空并设置文件描述符集合。

596-602：计时器初始化，用于记录重启时间



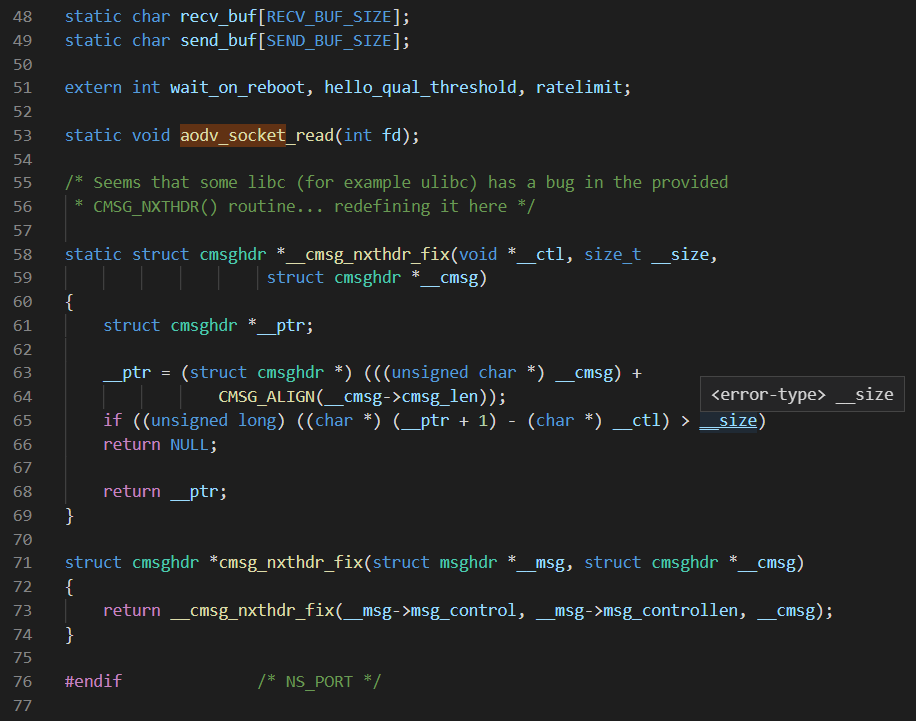
626-635：遍历结构体描述符集合如果有不为空则调用该结构体的函数指针回调函数，用于反复读取事件队列，判断是HELLO、RREQ、RREP的消息类型，并给予相应的操作，知道事件为NULL。

## main.c内部其他函数

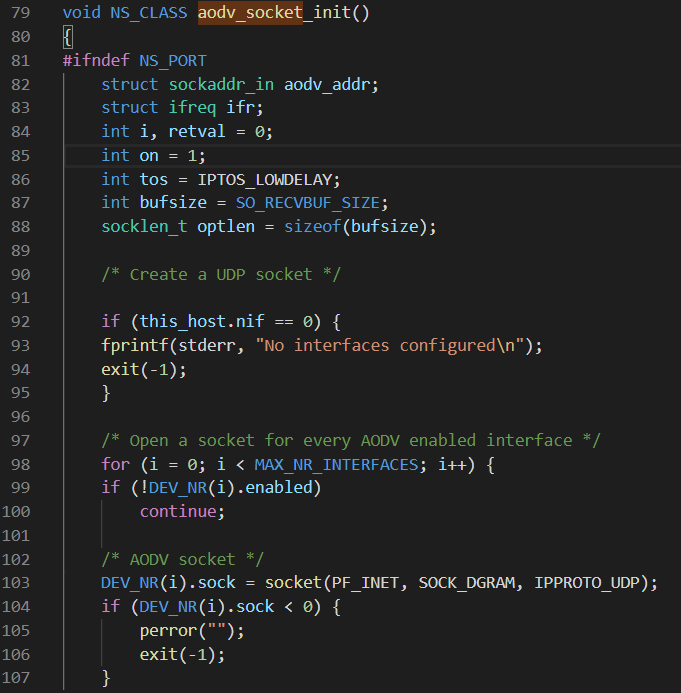
|  |  |  |
| --- | --- | --- |
| 函数名 | 返回值 | 作用 |
| clearup() | void | 清空路由表、套接字等数据结构和模块 |
| usage(int status) | void | 打印所有命令行参数及其作用 |
| set\_kernel\_option | int | 设置内核选项，在所有接口上禁用ICMP重定向 |
| find\_default\_gw() | int | 寻找默认网关 |
| get\_if\_info(char\*ifname,int type) | struct sockaddr\_in\* | 根据名字和类型，寻找相应的接口返回其信息 |
| attach\_call\_back\_func(int fd,callback\_func\_t func) | int | 增加一个callback数据结构元素，包括一个描述符和一个函数 |
| load\_modules() | void | 装载内核模块，如果模块在当前目录中则使用，否则调用modprobe |
| remove\_modules() | void | 删除某个模块 |
| host\_init() | void | 初始化某个端口 |
| signal\_handler(int type) | void | 处理各种信号 |
| ========== |  |  |

## aodv\_socket.c

该文件主要定义了aodv的udp套接字，用于接收和转发信息，下面将具体分析其架构。



48-51：定义接收数组、发送数组，用于开辟缓冲区为套接字使用

58-69：它声明了一个cmshdr结构 ，赋值为\_\_cmsg指针偏移一个长度， 在比较一下这个偏移后的指针和传进来的ctl指针差了多少，要是差的小于size就修复成了，差的太大就修复不成

79-106：遍历主机信息结构体如下

struct host\_info {

    u\_int32\_t seqno;        /\* Sequence number \*/

    struct timeval bcast\_time;  /\* The time of the last broadcast msg sent \*/

    struct timeval fwd\_time;    /\* The time a data packet was last forwarded \*/

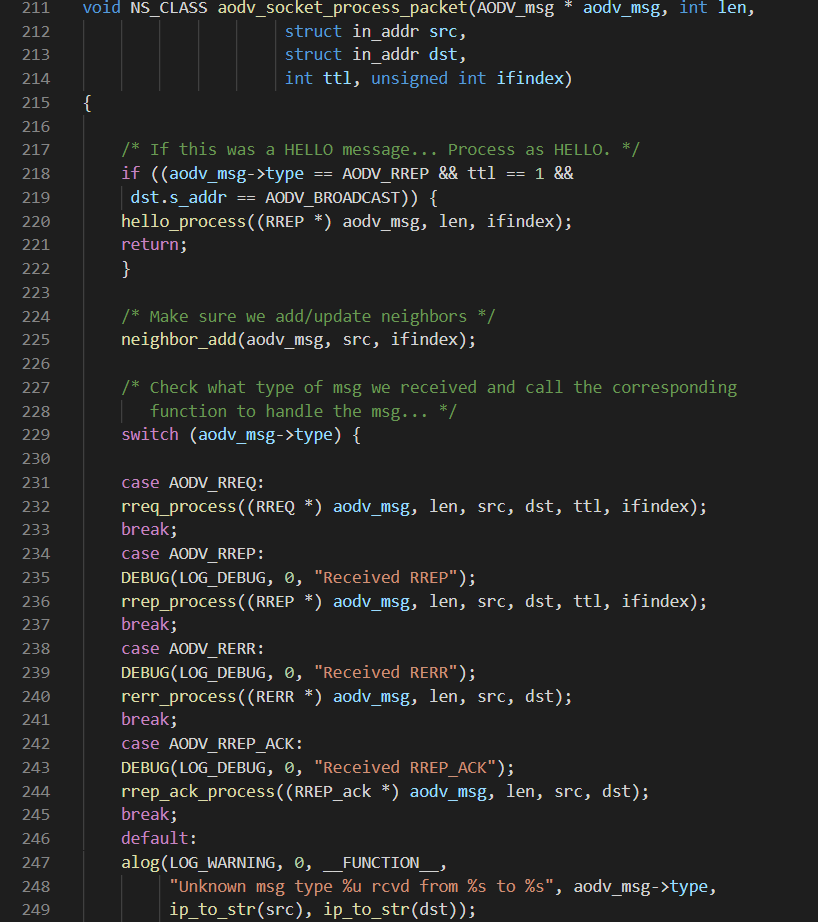
    u\_int32\_t rreq\_id;      /\* RREQ id \*/

    int nif;            /\* Number of interfaces to broadcast on \*/

        struct dev\_info devs[MAX\_NR\_INTERFACES+1]; /\* Add +1 for returning as "error" in ifindex2devindex. \*/

};

如果存在该结构体，则为其创建一个套接字（PF\_INET），并绑定监听（基本socket操作）

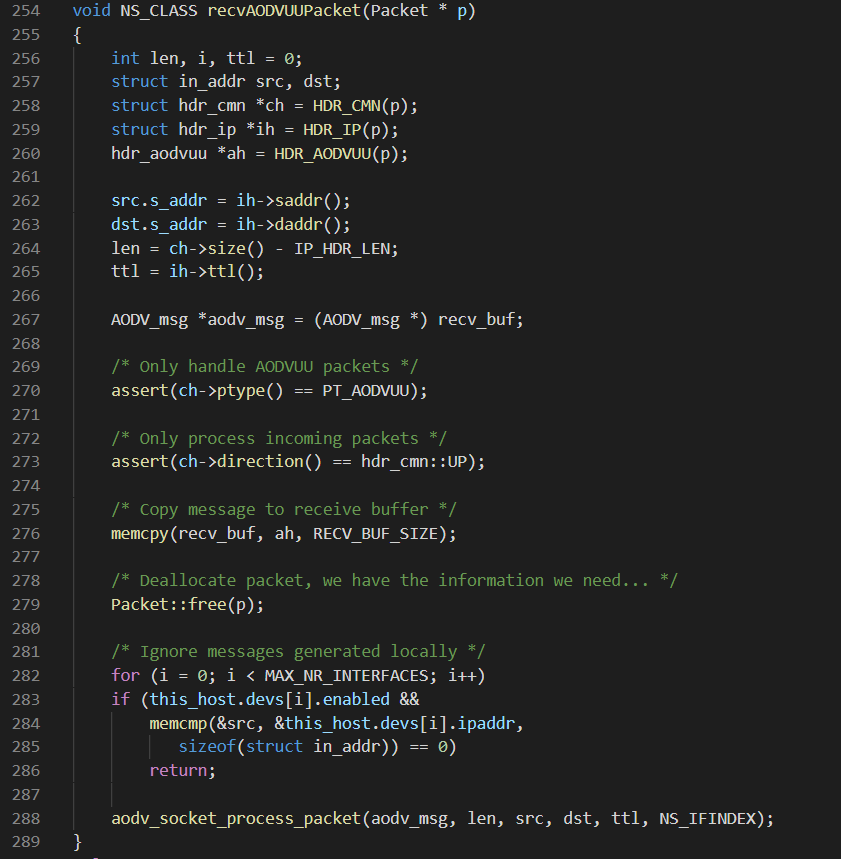


211-215：adov\_socket\_process\_packet函数，用于处理接收到的数据包，该函数参数为数据包结构体（AODV\_msg）、结构体长度、源节点套接字结构体、目的节点套接字结构体、TTL、以及传播参数。

218-222：如果该信息是广播的aodv\_hello消息，则传递给hello\_process处理

225：把该包和来源传递给neighbor\_add函数处理，由它决定是丢弃还是加进路由表中。

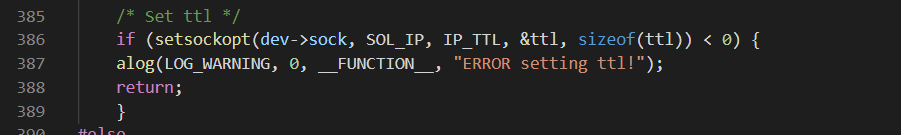
229-250：使用switch判断该消息的具体消息类型，用DEBUG记录于日志中，并交由各自消息的处理函数处理。



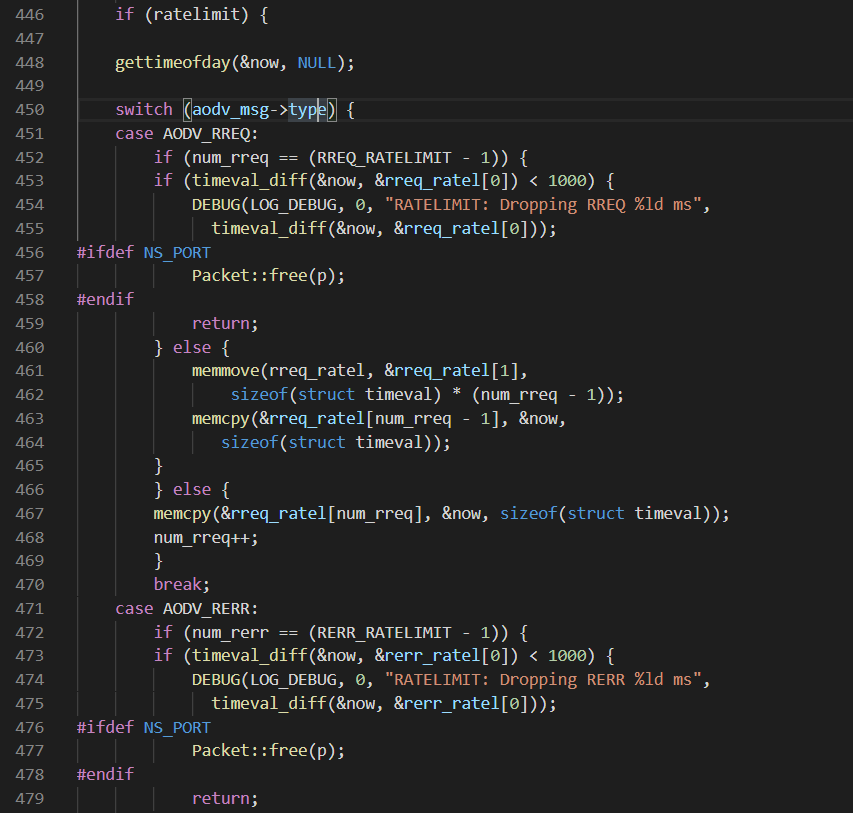
254-267：recvAODVUUPacket函数用于在上一个函数作用前对包的预处理，把包打开取出IP地址、TTL等信息。

270-273：检查收到的包是不是AODVUU，如果不是终止程序运行、检查包中的目的地址，只处理发给自己的包，如果不是终止运行。

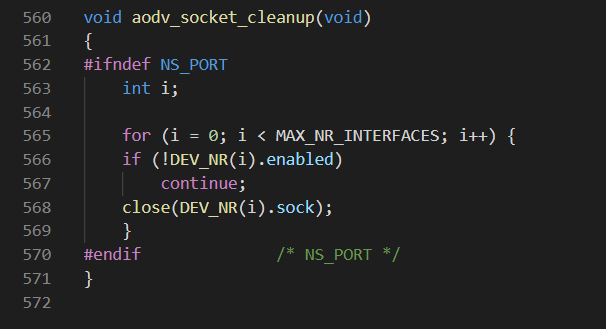
276-286：把数据拷贝到缓冲区，处理完毕后交由上个函数运行



386-390：设置套接字选项，规定往返时间



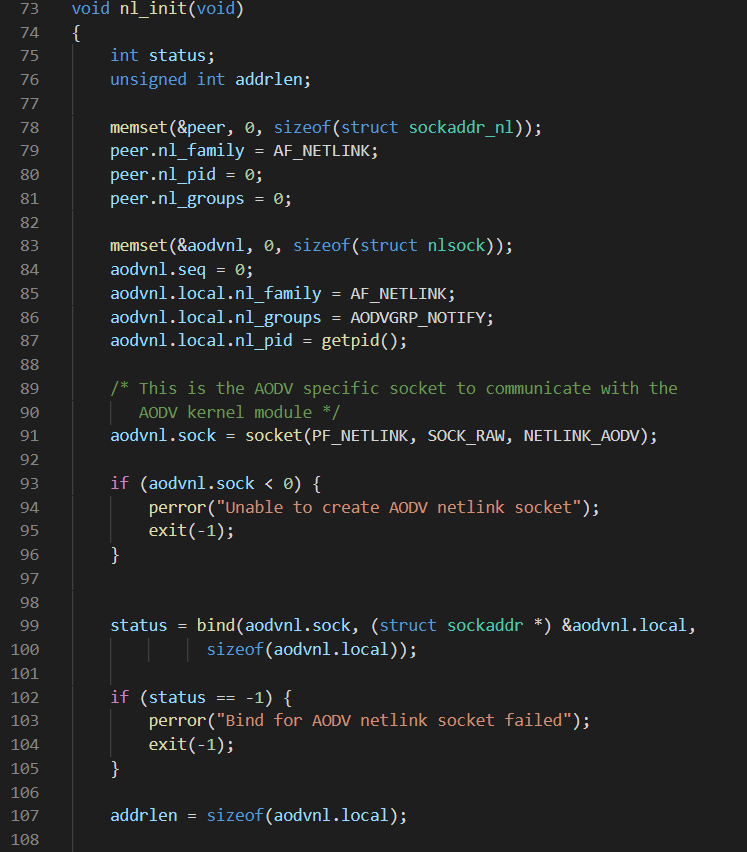
446-480：检查数据包是否有时间限制，如果有则分消息类型检查数据包是否在时间限制内，如果是则正常处理，如果不是则丢掉。



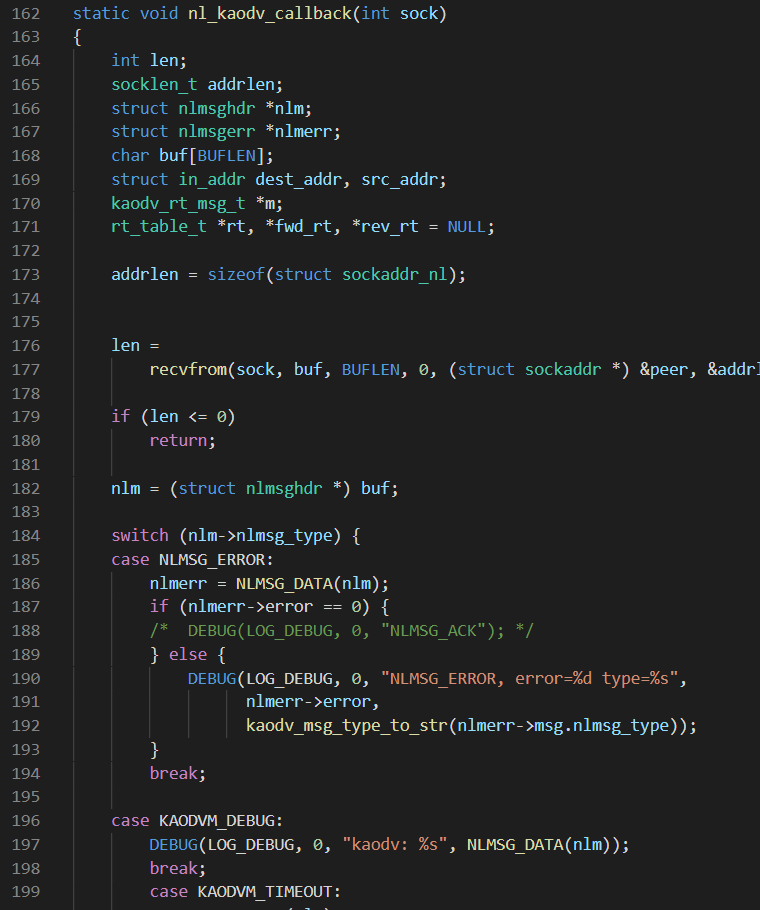
560-571：socket关闭连接

## nl.c

nl.c是aodv专用套接字，除了定义了一些aodv协议专用数据结构（如sockaddr\_nl、nl\_network等）之外，与正常套接字差别不大，我在这里只对其做简要分析。



该套接字初始化函数与上面的类似，这里不作介绍。



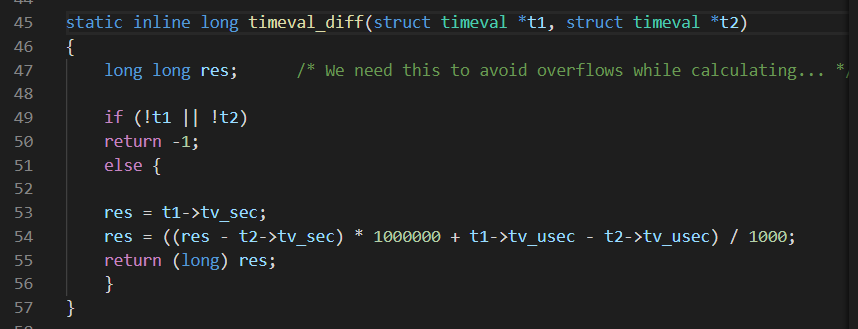
162-199：与LINUX内核交互，根据内核的不同命令执行相应的回调函数。

## 2.4.2 nl模块其余函数简介

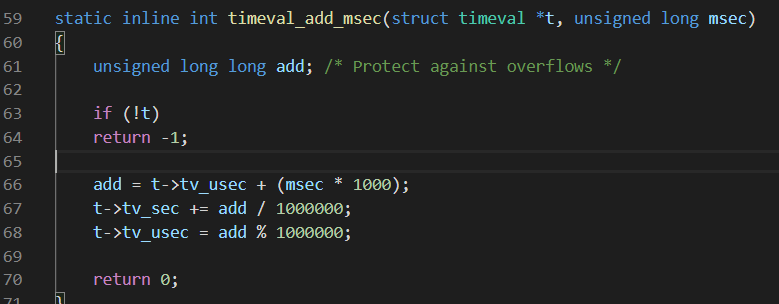
|  |  |  |
| --- | --- | --- |
| 函数名 | 返回值 | 作用 |
| nl\_cleanup(void) | void |  |
| nl\_rt\_callback(int sock) | void | 据内核套接字中收到的不同命令，执行不同的回调函数 |
| prefix\_length(int family, void \*nm) | int | 根据子网掩码计算出前缀长度 |
| nl\_send(struct nlsock \*nl, struct nlmsghdr \*n) | int | 发送数据包至内核 |
| nl\_kern\_route() | int | 操作路由表 |
| nl\_send\_add\_route\_msg | int | 添加路由信息（目的地址、下一条、TTL、flag、路由状态套接字、网络接口等信息） |
| nl\_send\_no\_route\_found\_msg | int | 向内核发送一个消息，标记到该目的地的路由信息为无法找到 |
| nl\_send\_del\_route\_msg(struct in\_addr dest, struct in\_addr next\_hop, int metric) | int | 向内核发送一条消息，要求删除一条路由条目 |
| int nl\_send\_conf\_msg(void) | int | 向内核发送一条消息，要求进行设置 |

## timer\_queue.c

用于定义aodv计时器，计算微秒级的时间差等功能函数模块。



45-57：计算传入两个timeval的时间差

  
66：将毫秒转化成微妙

67：将add时间中的整数秒取出赋给tv\_sec

68：将add中的微妙赋值给usec

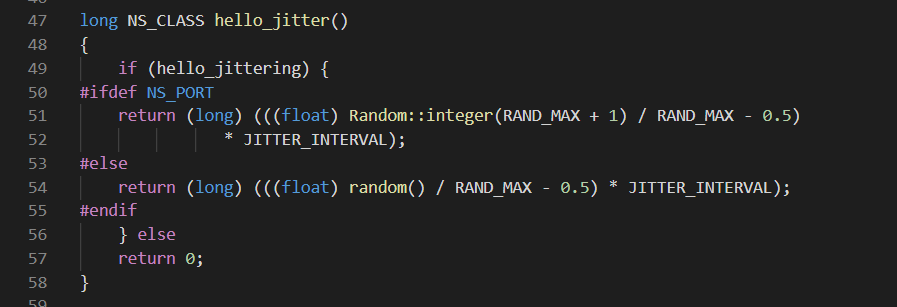
## 2.5.2 time\_queue.c主要其他函数用途

|  |  |  |
| --- | --- | --- |
| 函数名称 | 返回值 | 用途 |
| timer\_queue\_init() | void | 结构体初始化赋值 |
| timer\_timeout(struct timer \*now) | void | 从TQ中删除过期计时器并将其赋值给expTQ，在expTQ安全地从头部删除该计时器 |
| timer\_add(struct timer\*t) | void | 检查计数器是否正常并且找到合适的位置加入新的计数器 |
| timer\_remove(struct timer\*t) | int | 删除所选择的计数器 |
| timer\_timeout\_now | int | 获取计数器现在的数值，成功返回1不成功返回-1 |
| timer\_set\_timeout(struct timer\*t,long mesc) | void | 如果原来有该计数器存在于队列，删除它，设置新的计数器值 |
| timer\_left(struct timer\*t) | long | 返回剩余时间 |
| timer\_age\_queue() | struct timeterval\* | 返回计数器队列对头 |

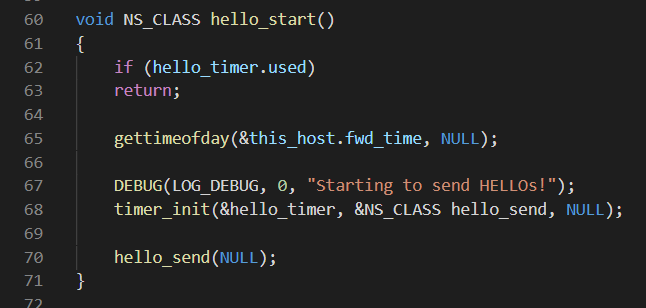
## aodv\_hello.c

hello消息的定义与操作

## 2.6.1 初始化hello消息



47-59：返回一个随机时长



60-71：hello消息发送，如果hello计时器被占用则不发送，否则启动计时器发送广播hello报文。

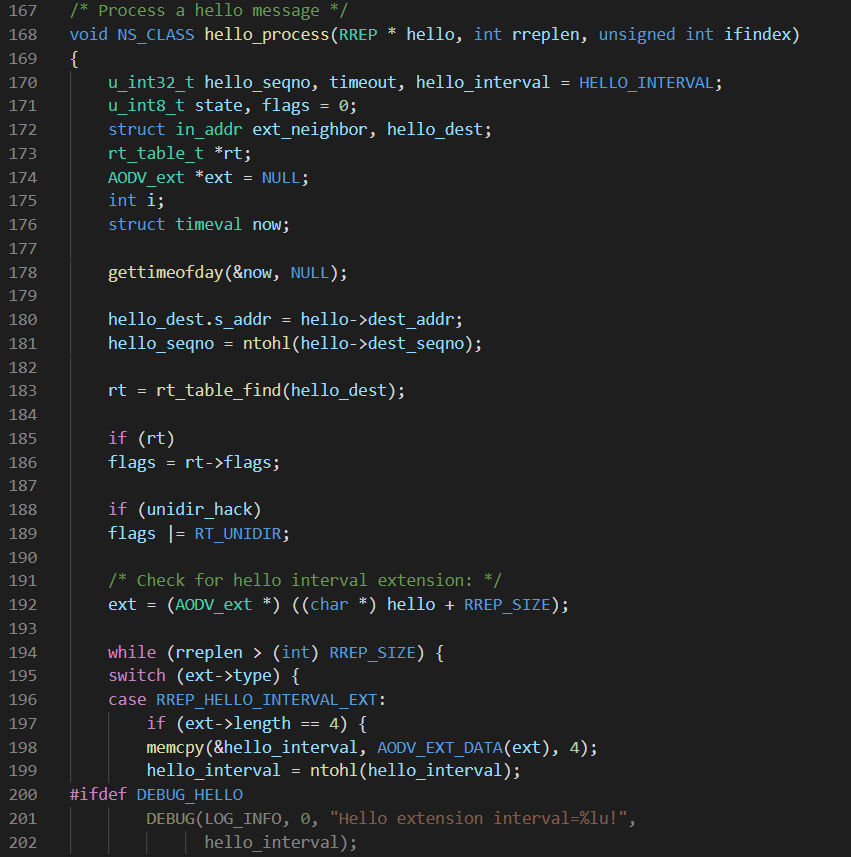


80-96：消息初始化，并检查当前和最后一条hello消息计时器是否超时，如果超时则停止转发hello消息，触发hello\_stop()

99-100：获取当前时间与广播最后一条消息的时间差

104-116：如果时间差大于预定的时间差则创建rrep消息（内容赋值为源节点ip地址（DEV\_NR(i).ipaddr），目的节点ip，序列号，允许发送时长）并广播。

## 2.6.2 处理hello消息

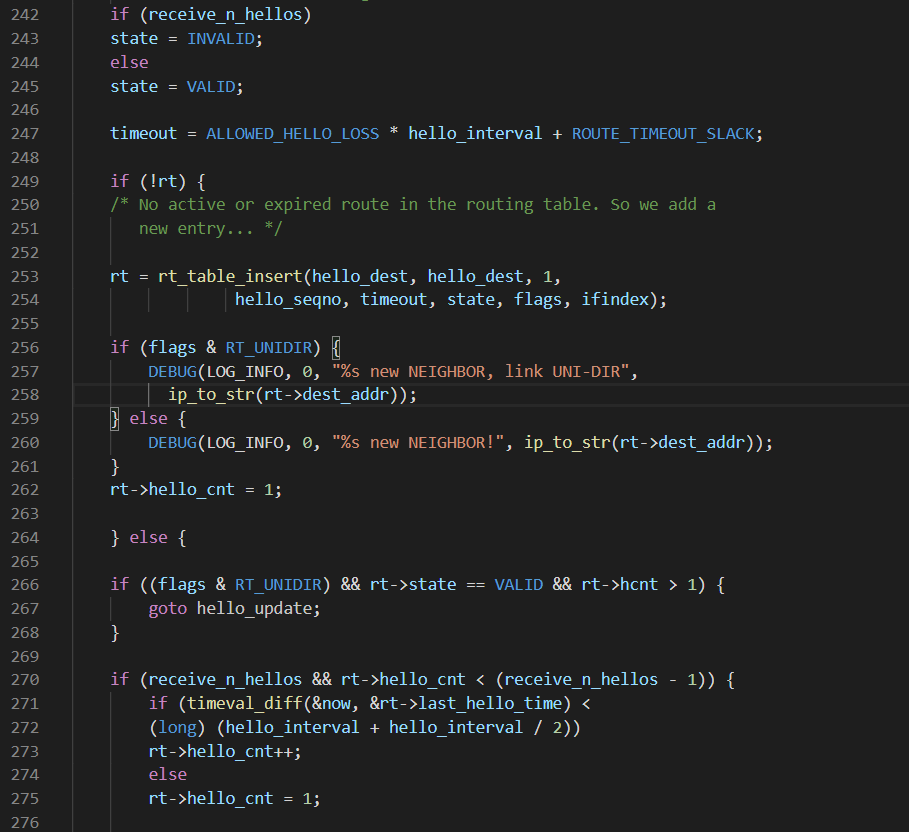


169-183：初始化路由表、hello序列号、超时时间、IP地址等变量信息

185-192：检查路由表并开始转发hello消息、初始化AODV报文头ext



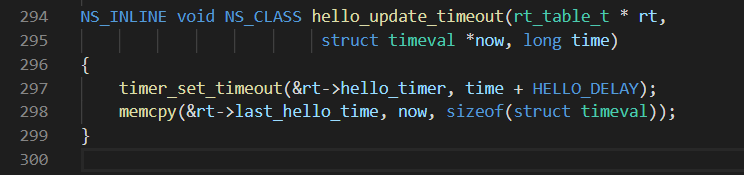
检查ext报文类型，判断其实REEO+HELLO\_INTERVAL\_EXT类型还是RREP\_HELLO\_NEIGHBOR\_SET\_EXT类型。（这段没看懂）



241-244：只有当邻居节点收到三个连续的hello消息时才有效，根据此设置”INVALID”(不可用)或”VALID”(可用)。

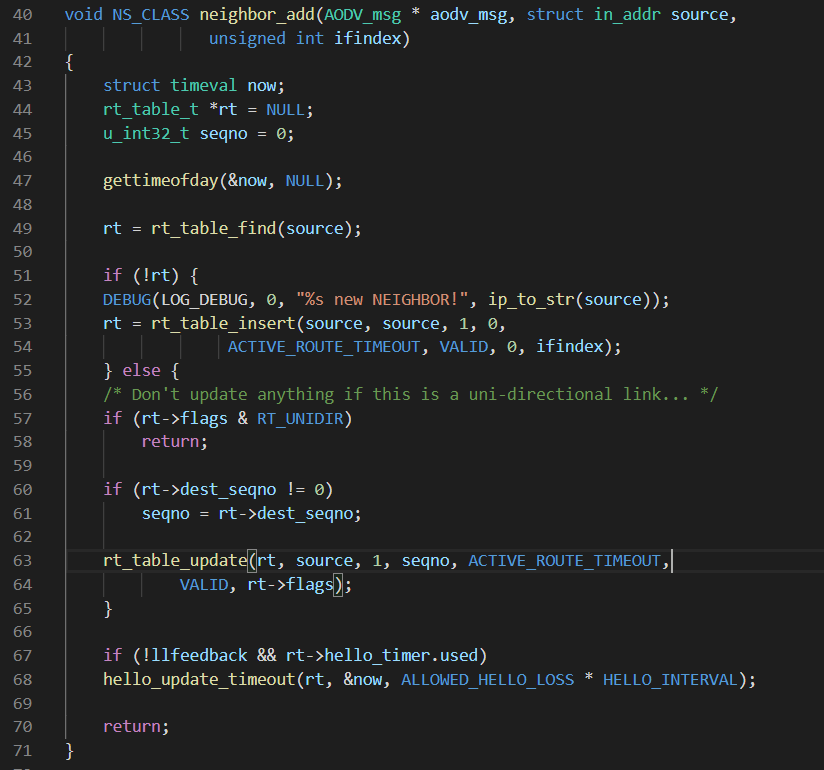
248-261：当一个结点从邻近结点收到了一个hello消息，该节点应该确保它于此邻近结点有一条活动的路由。当在路由表中没有活跃的或者过期的路由表项时，我们应该增加一个新的表项。rt\_table\_insert()函数输入一系列参数即创建了表项。

265-276：当路由表中状态为可用而且到达目的节点的跳数大于1则更新hello消息，同时该路径的生命周期应该增加。

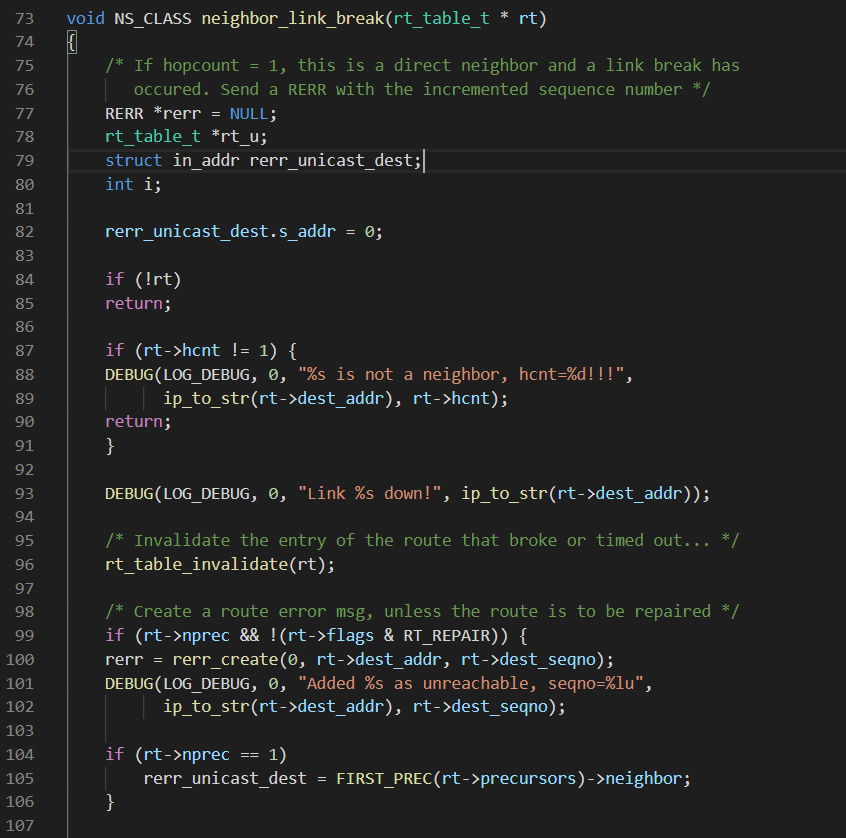


294-300：宏定义HELLO\_DELAY，因为网络上的处理延迟，所以我们允许hello消息存在延迟，这里我们设置为50。

## 2.7 邻节点操作



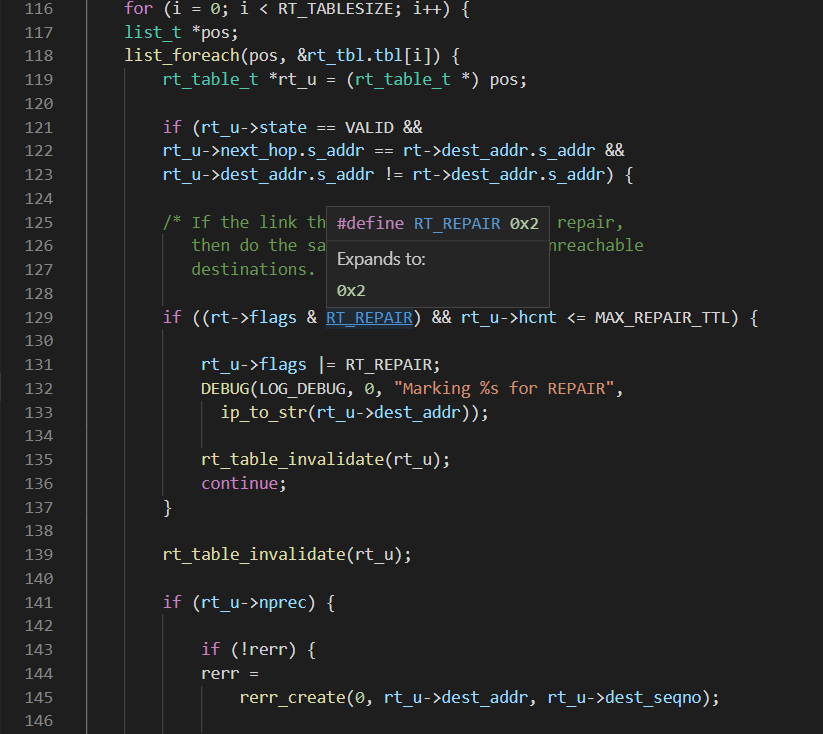
51-70：如果是新节点，则在路由表中增加一条指向自己的记录，否则更新路由表中的序列号。



77-90：判断检验的节点是否跳数为1，如果不是，则不是相邻的节点不需要断开。

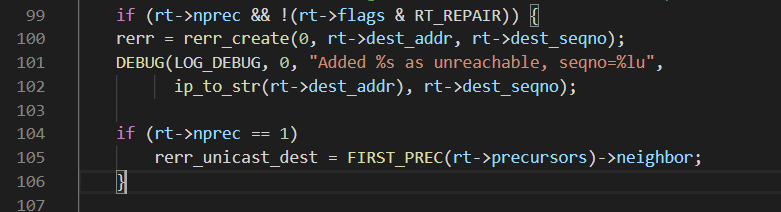
96：使中断或超时的路由条目无效

90-102：如果在时间内没有被修复，则产生错误消息

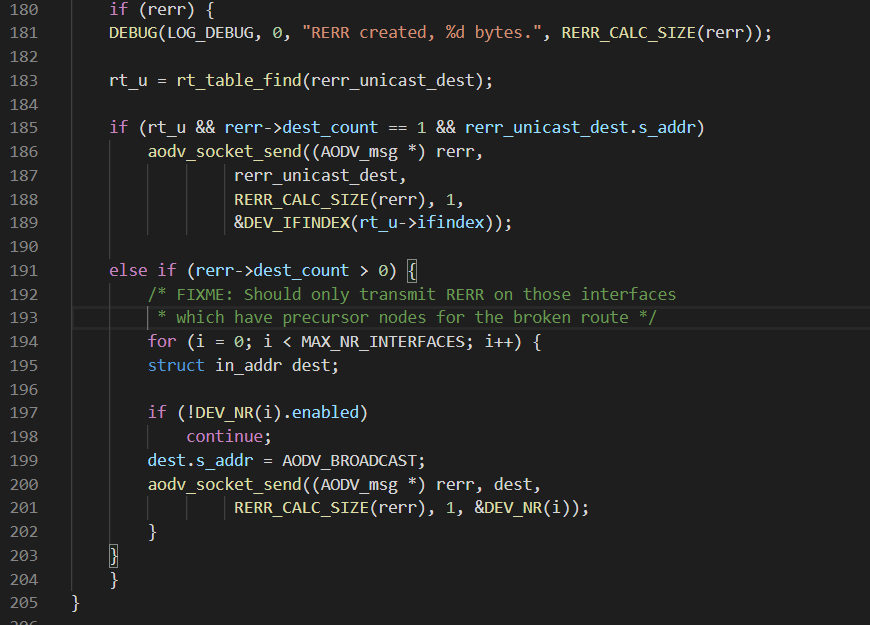


116-123：检查路由表中是否有目的地（目的地）为下一跳的条目。 由于dest已关闭，因此无法访问这些条目（目的地）。 因此，它们也应包含在RERR中。

129-146：如果该链路被标记正在修复，对其他节点也如此检查



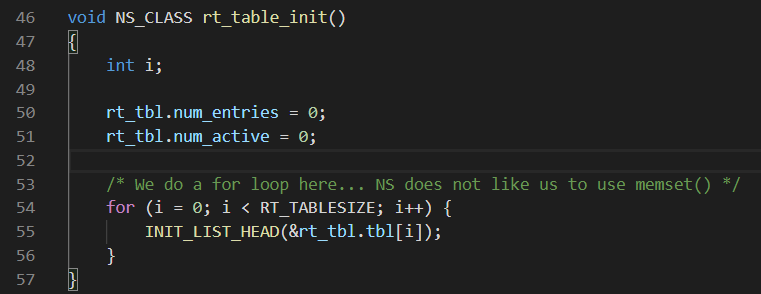
99-106：如果无法到达的链路没有被修复，则向该节点邻居单播RERR消息。



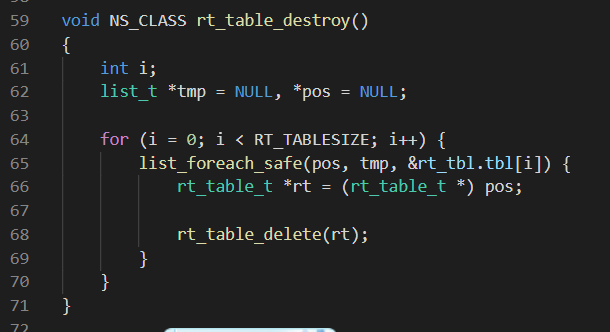
180-201：向之前判断好的已损坏的、没在修复状态的、存在前驱节点的邻接点发送RERR消息。

## 路由表操作- routing\_table.c

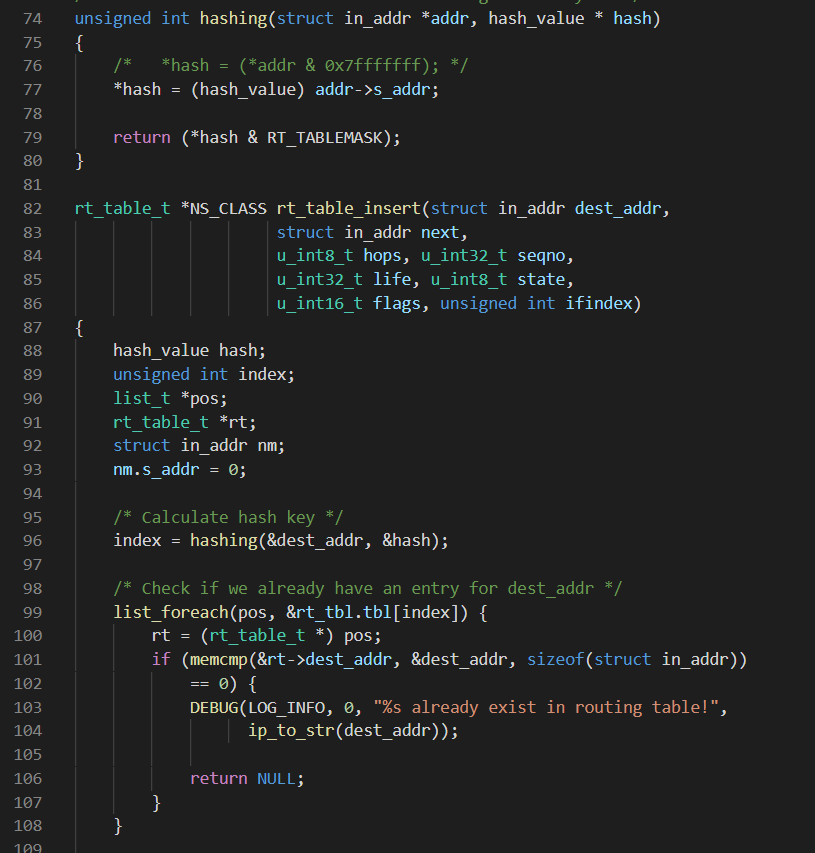
AODV最核心的部分就是对路由表的增删改查，前面的所有模块几乎都是为这一模块所服务，下面我将具体介绍路由表的相关操作。



46-57：rt\_table\_init()：初始化路由表



59-70：rt\_table\_destroy()：销毁路由表

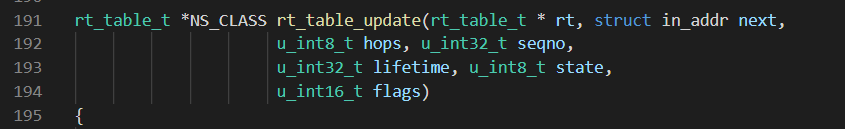


74-80：hashing(struct in\_addr\*addr,hash\_value\*hash)：计算哈希值

82-109：rt\_table\_insert(struct in\_addr dest\_addr, struct in\_addr next u\_int8\_t hops, u\_int32\_t seqno,

                     u\_int32\_t life, u\_int8\_t state,

                     u\_int16\_t flags, unsigned int ifindex)：路由表插入

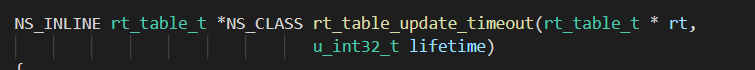


191-195：rt\_table\_update(rt\_table\_t \* rt, struct in\_addr next,

                     u\_int8\_t hops, u\_int32\_t seqno,

                     u\_int32\_t lifetime, u\_int8\_t state,

                     u\_int16\_t flags):更新已有表项信息



函数六：rt\_table\_update\_timeout(rt\_table\_t \* rt,

                               u\_int32\_t lifetime)更新路由表计时器



函数七：rt\_table\_update\_route\_timeouts(rt\_table\_t \* fwd\_rt,

rt\_table\_t \* rev\_rt)更新超时实践，用于回应发出或者收到的数据包，所有更新的事件都要顺着路径返回，因为源节点和目的节点对于数据回应的需求是对称的



函数八：tr\_table\_find(struct in\_addr desk\_addr)：在路由表中查询是不是有该目的节点的ip地址的表项，如果有返回该表索引



函数九：rt\_table\_find\_gateway()：寻找本机默认网关



函数十：检测路线是否到期，如果到期将之删除

# 3 总结与心得