**网络协议栈分析与设计课程大作业**

|  |
| --- |
| **AODV路由协议代码分析** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 学号 | 姓名 | 班级 | 负责模块 | 成绩 | | 201692356 | 陈一飞 | 软网1603 | 1.阅读rfc文档  2.阅读二分之一源代码  3.搜集资料，分析协议架构  4.撰写协议功能分析  5.整理文档格式 |  | | 201692187 | 韩雨生 | 软网1603 | 1.阅读rfc文档  2.阅读二分之一源代码  3.搜集资料，撰写代码分析  4.制作协议相关图表  5.审阅校对 |  | |

目录

[第一章 引言 4](#_Toc23734)

[1.1 定义 4](#_Toc5751)

[1.2 特点 5](#_Toc999)

[1.3 实现技术 6](#_Toc23916)

[1.4 路由表字段 6](#_Toc21903)

[1.5 相关的路由协议 7](#_Toc24705)

[第二章 代码介绍 8](#_Toc8520)

[2.1 文件介绍 8](#_Toc18184)

[第三章　数据结构及相关操作 9](#_Toc14012)

[3.1 struct list\_t 9](#_Toc14649)

[3.1.1 list\_t 结构 9](#_Toc8102)

[3.1.2 list\_t相关操作函数 9](#_Toc15007)

[3.2 struct timer++ 10](#_Toc30499)

[3.2.1 timer 结构 10](#_Toc28858)

[3.2.2 timer 相关函数 10](#_Toc12351)

[3.3 struct precursor 11](#_Toc32710)

[3.4 struct routing\_table++ 11](#_Toc28185)

[3.4.1 routing\_table 结构 11](#_Toc14041)

[3.4.2 routing\_table 相关函数 12](#_Toc25120)

[3.5 struct routing\_table++ 14](#_Toc19878)

[3.6 struct RREP 14](#_Toc2352)

[3.6.1 RREP 结构 14](#_Toc18861)

[3.6.2 RREP 相关函数 15](#_Toc9880)

[3.6.2 RREP 相关知识 16](#_Toc3490)

[3.7 struct RREQ 17](#_Toc5200)

[3.7.1 RREQ 结构 17](#_Toc27983)

[3.7.2 RREQ 相关函数 18](#_Toc31212)

[3.7.3 RREQ 相关知识 19](#_Toc12603)

[3.8 Struct RRER 19](#_Toc18442)

[第四章 AODV 路由协议的实现 22](#_Toc4318)

[4.1 main.c 22](#_Toc1100)

[4.1.1 main函数 22](#_Toc7984)

[4.1.2 main.c中其他函数 27](#_Toc4648)

[4.2 AODV路由发现 27](#_Toc17606)

[4.2.1 广播RREQ路由请求帧 28](#_Toc18182)

[4.2.2 中间节点更新各自到源节点的路由表 33](#_Toc31721)

[4.2.3 中间节点处理RREQ消息 36](#_Toc19593)

[4.2.4 产生RREP 44](#_Toc26178)

[4.2.5 RREP单播至源节点 47](#_Toc10746)

[4.2.6 收到RREP 48](#_Toc20757)

[4.3 AODV路由维护 51](#_Toc22060)

[4.3.1 Hello消息 51](#_Toc9248)

[4.3.2 发送Hello消息 51](#_Toc32078)

[4.3.3 链路失效判断 54](#_Toc22675)

[4.3.4 局部修复 55](#_Toc17034)

[4.3.5 修复超时，发送RRER路由错误信息 57](#_Toc16581)

[4.3.6 中间节点处理RRER 57](#_Toc17251)

[4.4 AODV路由信息新旧判断 59](#_Toc28302)

[4.4.1 源节点更新序列号 60](#_Toc4905)

[4.4.2 目的节点更新序列号 60](#_Toc13680)

[4.4.3 对比序列号判断路由消息的新旧 60](#_Toc22279)

[4.5 拥塞控制 62](#_Toc28397)

[第五章 总结 64](#_Toc12142)

# 引言

无线自组网按需平面距离向量路由协议（Ad hoc On-Demand Distance Vector Routing，AODV）是应用于无线随意网络（也称作无线Ad hoc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Ad Hoc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，避免了传统的距离向量协议中会出现的很多问题。

## 1.1 定义

AODV 是一种源驱动路由协议。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，aodv 详细的工作流程图见图1.1。

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。

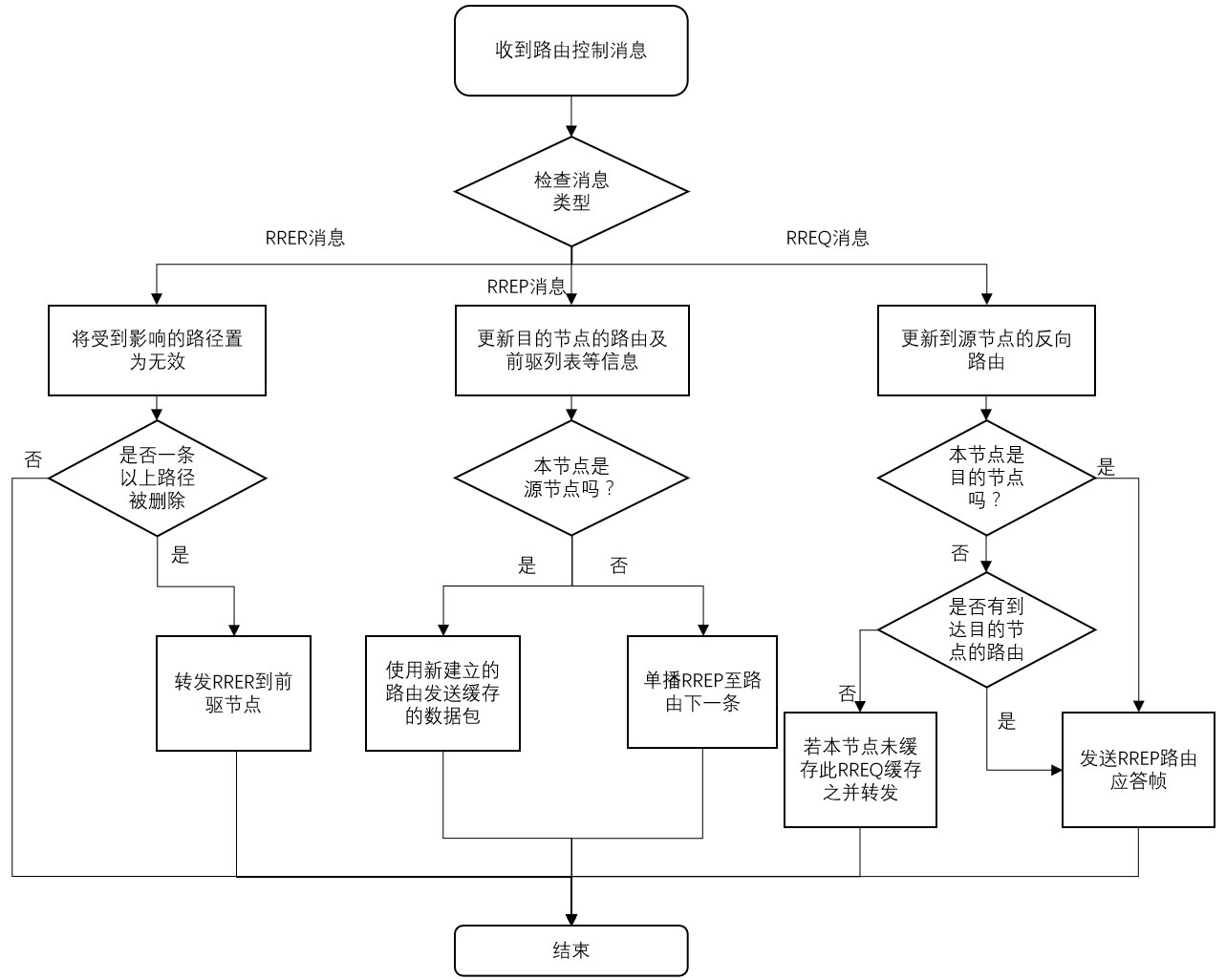


图1.1 节点收到各数据帧后执行的功能

## 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属于网络层协议。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。

## 1.3 实现技术

它是反应式路由协议，也就是说当向目的节点发送包时，源节点才在网络中发起路由查找过程，找到相应的路由。相反的，很多普通的因特网路由协议都是先验式的，也就是说它们查找路由是不依赖于路径上的节点是否要发包，而是每个节点维护一张包含到达其它节点的路由信息的路由表。节点间通过周期性的交换路由信息来不断更新自身的路由表，以便能够及时的反映网络拓扑结构和变化，以维护一致的、及时的、准确的路由信息。正如协议的名字所示，无线自组网按需平面距离矢量路由协议是一种平面距离矢量路由协议。

在AODV中，整个网络都是静止的除非有连接建立的需求。这就是说一个网络节点要建立连接时才广播一个连接建立的请求。其他的AODV节点转发这个请求消息，并记录源节点，和回到源节点的临时路由。当接收连接请求的节点知道到达目的节点的路由时，就把这个路由信息按照先前记录的回到源节点的临时路由发回源节点。于是源节点就开始使用这个经由其他节点并且有最短跳数的路由。当链路断掉，路由错误就被回送给源节点，于是源节点就重新发起路由查找的过程。

大多数协议的复杂性在于为了保证网络性能而减少消息数量。例如，每个路由请求都会有一个序号，节点使用这个序号以避免它们重复转发这个路由请求。路由请求有一个“生存时间”数，这将减少他们被重传的次数。还有就是如果路由请求失败，其他的路由请求将会在先前的路由请求消息超时后的两倍的“生存时间”之后，才被发送。

## 1.4 路由表字段

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)目的节点序列号有效标志位。

(4)下一跳节点IP地址。

(5)本节点到达目的节点的跳数。

(6)前驱节点列表（precursor list）。

(7)生存时间（路由失效或删除时间）。

(8)网络层接口。

(9)其他的状态和路由标志位。

其特征如下：

(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)源节点和目的节点都维护各自的序列号。

(3)如何管理序列号是提高路由建立和维护的关键。

I.序列号是用来标识路由信息新旧程度（freshness）的。

II.源节点发起路由请求 RREQ，或者目的节点返回路由应答RREP，都要更新各自的序列号。

III.其他节点（中间节点）依据序列号的大小判断路由的新旧。

## 1.5 相关的路由协议

AODV对在这方面有多种解决方法。还有一种路由协议是动态源路由协议(DSR)，这个路由协议充分最优化网络的通信量。另外就是优化的链路状态路由协议(OLSR)也是解决这方面问题。OLSR不断地收集节点之间能相互通信的数据，并对每个节点保持一个最优化的路由表。所以连接可以很快的建立。但是OLSR是一个相对比较大而且复杂，它要求大型复杂的计算机、很大的内存和计算。同时频繁进行网络其他节点的发现过程是一个巨大的负担。对于其他可选择的方法可以参照无线自组网协议列表。

# 第二章 代码介绍

## 2.1 文件介绍

AODV协议是由C语言实现的，头文件加上源文件一共44个，下表给出了主要的文件并对其主要实现的功能进行了说明：

|  |  |
| --- | --- |
| 文件名 | 说明 |
| aodv\_hello.[ch]  aodv\_neighbor.[ch]  aodv\_rerr.[ch]  aodv\_rrep.[ch]  aodv\_rreq.[ch]  aodv\_socket.[ch]  aodv\_timeout.[ch]  debug.[ch]  defs.h  endian.c  list.[ch]  llf.[ch]  locality.[ch]  main.c  nl.[ch]  params.h  routing\_table.[ch]  seek\_list.[ch]  time\_queue.[ch]  ChangeLog  GPL  Makefile  README  README.ns  TAGS  TODO | Hello消息的发送接收和处理。  邻居节点添加，链路断开的处理。  关于rerr(链路错误消息)的发送接收和处理。  路由回应消息  路由请求消息  AODV套接字发送接收以及处理  发现以及处理超时  记录日志  宏定义  补充某些缺乏<endian.h>的系统  链表操作  链路层反馈  寻找目的地方位  AODV协议主函数  AODV协议专用套接字  常用参数定义  路由表  链表操作  定时器有关操作  日志记录文件  非正式公共授权说明  编译配置命令  帮助文档  帮助文档2  实现函数原型、数据结构体声明等快速跳转  对本协议未来展望 |

表2.1 AODV路由协议主要文件

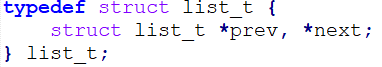
# 第三章　数据结构及相关操作

## struct list\_t

### 3.1.1 list\_t 结构

struct list\_t 如下截图所示：

list.h



list.h

struct list\_t拥有两个指向list\_t的指针，是一个典型的双向链表节点结构。该结构体用于构建双向循环链表。

### 3.1.2 list\_t相关操作函数

在list.c文件中，提供了如下表所示的函数。

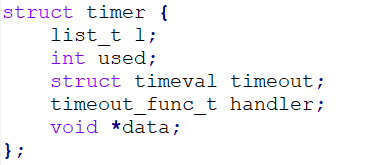
|  |  |
| --- | --- |
| 函数原型 | 功能 |
| static inline int listelm\_detach(list\_t \* prev, list\_t \* next); | 删除某个位置的节点。 |
| static inline int listelm\_add(list\_t \* le, list\_t \* prev, list\_t \* next); | 在两节点之间添加一个新的节点le。 |
| int list\_add(list\_t \* head, list\_t \* le); | 将le插入head节点与head后一节点之间。 |
| int list\_add\_tail(list\_t \* head, list\_t \* le); | 将le插入head前一节点与head节点之间。 |
| int list\_detach(list\_t \* le); | 删除某节点。 |

## struct timer++

### 3.2.1 timer 结构

struct timer 如下截图所示：

timer\_queue.h



timer\_queue.h

struct timer结构共由五部分组成。timer结构在AODV协议中多次出现，用于定义定时器以监视事件。

list\_t l； 用于构建timer链表。

int used； 用于标识该timer是否被使用。

timeval timeout； 用于标识超时时间的计时器。

timeout\_func\_t handler； 是一个指向返回值为void \*的函数指针。

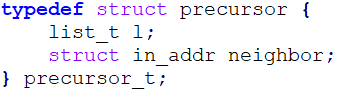
void \*data； 无类型指针。

### 3.2.2 timer 相关函数

在timer\_queue.c，timer\_queue.h文件中，提供了如下表所示的函数。

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| static inline long timeval\_diff  (struct timeval \*t1, struct timeval \*t2); | 返回t1与t2时间的差值。 |
| static inline int timeval\_add\_msec  (struct timeval \*t, unsigned long msec); | 为计时器t增加时长msec的时间。 |
| int NS\_CLASS timer\_init  (struct timer \*t, timeout\_func\_t f, void \*data); | 用于初始化定时器。 |
| void NS\_CLASS timer\_timeout  (struct timeval \*now); | 当一个定时器超时时，调用此函数进行  后续的清理工作。 |
| NS\_STATIC void NS\_CLASS timer\_add  (struct timer \*t); | 在定时器队列中添加一个定时器。 |
| int NS\_CLASS timer\_remove(struct timer \*t); | 在定时器队列中删除一个定时器。 |
| int NS\_CLASS timer\_timeout\_now  (struct timer \*t); | 让此定时器t立即超时。 |
| void NS\_CLASS timer\_set\_timeout  (struct timer \*t, long msec); | 在定时器原来的时间上再加上msec长度的时间。 |
| long timer\_left(struct timer \*t); | 返回计时器t剩余的时间。 |
| struct timeval \*NS\_CLASS timer\_age\_queue(); | 返回定时器队列的生存时间。 |
| void NS\_CLASS printTQ(list\_t \* l); | 打印出定时器队列l中每个定时器的信  息。 |

## struct precursor



struct precursor由两部分组成。list\_t l用于构建链表。in\_addr neighbor用于表明邻居节点，即前驱表的ip地址。

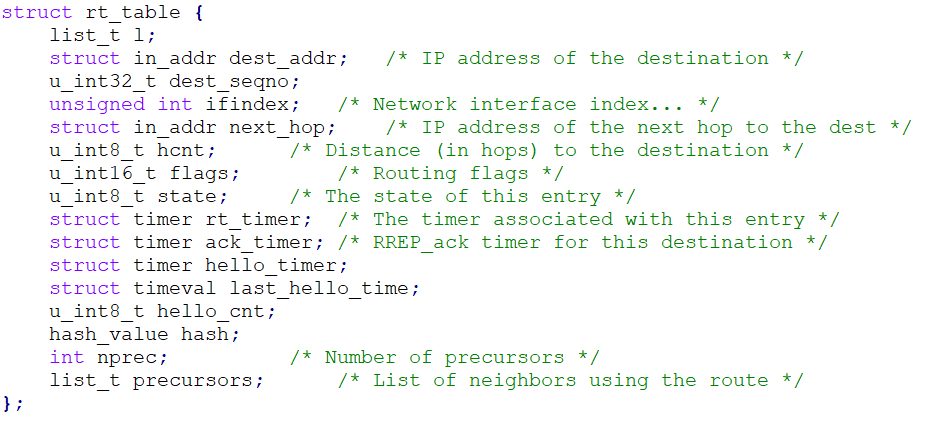
前驱表：在 AODV 路由表中，每一条目都有一个先驱表。表里的节点（先驱）可能会沿着这条路由转发数据包。当检测到下一跳断开时，这些先驱就会从本节点收到通知。路由表项里的先驱列表包含的是都是本节点的相邻节点，路由回复信息将会被发送到这些节点。

## struct routing\_table++

### 3.4.1 routing\_table 结构

struct routing\_table如下截图所示：

routing\_table.h



routing\_table.h

图3.1 routing\_table的代码定义

rt\_table是一项用于储存路由表项的结构体。具体如下：

list\_t l; 用于构建链表。

in\_addr dest\_addr; 目的地ip地址。

u\_int32\_t dest\_seqno; 目的地序列号。

unsigned int ifindex; 网络接口(索引)。

in\_addr next\_hop; 下一跳地址。

u\_int8\_t hcnt; 目的地距离，跳数。

u\_int8\_t state; 路由状态。

u\_int16\_t flags; 路由标记。

timer rt\_timer; 与这个入口关联的计时器

timer ack\_timer; 用于标识ack的计时器

timer hello\_timer; 用于标识hello的计时器

timeval last\_hello\_timer;上次使用该路由表项的时间。

hash\_value hash; 哈希值

list\_t precursors 前驱节点链表。

### 3.4.2 routing\_table 相关函数

在routing\_table.c，routing\_table.h文件中，提供了如下表所示的函数，用于对路由表进行操作。

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| void NS\_CLASS rt\_table\_init() | 初始化路由表。 |
| void NS\_CLASS rt\_table\_destroy() | 删除路由表。 |
| unsigned int hashing(struct in\_addr \*addr, hash\_value \* hash) | 根据IP地址，计算哈希值。 |
| rt\_table\_t \*NS\_CLASS 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) | 根据参数目的地地址，下一跳，跳数，目的地序列号，生命期，路由状态，正确标记，网络接口，增加一条路由表项。 |
| rt\_table\_t \*NS\_CLASS 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) | 若传来的路由表项rt已过期，又再次活跃，则更新路由表。 |
| NS\_INLINE rt\_table\_t \*NS\_CLASS rt\_table\_update\_timeout (rt\_table\_t \* rt, u\_int32\_t lifetime) | 检查并更新路由表项中计时器。 |
| void NS\_CLASS rt\_table\_update\_route\_timeouts  (rt\_table\_t \* fwd\_rt, rt\_table\_t \* rev\_rt) | 更新输入(rec\_vt)或输出(fwd\_rt)包路由时的定时器。 |
| rt\_table\_t \*NS\_CLASS rt\_table\_find(struct in\_addr dest\_addr) | 根据目的地地址，寻找路由表项。 |
| rt\_table\_t \*NS\_CLASS rt\_table\_find\_gateway() | 寻找本机的默认网关。 |
| int NS\_CLASS rt\_table\_update\_inet\_rt  (rt\_table\_t \* gw, u\_int32\_t life) | 设置在默认状态下，所有包的下一跳均传给默认网关。 |
| int NS\_CLASS rt\_table\_invalidate(rt\_table\_t \* rt) | 若路由表项超时，则将其无效化。 |
| void NS\_CLASS rt\_table\_delete(rt\_table\_t \* rt) | 删除一个路由表项。 |
| void NS\_CLASS precursor\_add(rt\_table\_t \* rt, struct in\_addr addr) | 在某个路由表项的前驱表中，添加一个节点。 |
| void NS\_CLASS precursor\_remove(rt\_table\_t \* rt, struct in\_addr addr) | 在某个路由表项的前驱表中，删除一个节点。 |
| void precursor\_list\_destroy(rt\_table\_t \* rt) | 将某个路由表项的前驱表删除。 |

对这里相近的几个函数进行对比。

* rt\_table\_update\_timeout 和 rt\_table\_update\_route\_timeouts：

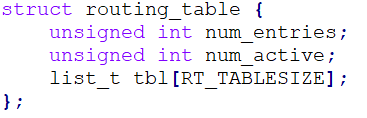
rt\_table\_update\_timeout 用来设置路由表中，某一个条目的生命期。而rt\_table\_update\_route\_timeouts 是当有包开始要路由时，要更新源到目的的路由条目的生命期，同时也要更新反向的线路的生命期。这是因为如果有数据包用到了这条路由，表明此路由目前为止还是有价值的，所以要更新其生命期。两个函数中的后者的函数体内部，要调用前者。

* rt\_table\_invalidate 和 rt\_table\_delete：

rt\_table\_invalidate 的作用是当某个路由表项过期，或者发生错误时将其标记为“无效”，并没有将其立即删除。而且以后也许也重用或者更新的可能。而rt\_table\_delete的作用是将某个路由表项直接删除，路由表中不再有它的信息。

## struct routing\_table++

routing\_table.h



routing\_table.h

图3.2 struct routing\_table的代码定义

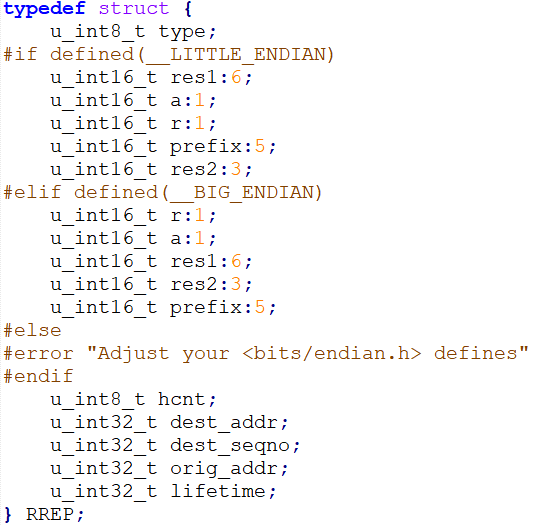
struct routing\_table用于记录本机路由表的相关信息。内含两个int类型的变量。num\_entries表征路由表中的路由表项总数。num\_active表征路由表中活跃的路由表项数。list\_t数组tbl用于存储本地的路由表项。

## struct RREP

### 3.6.1 RREP 结构

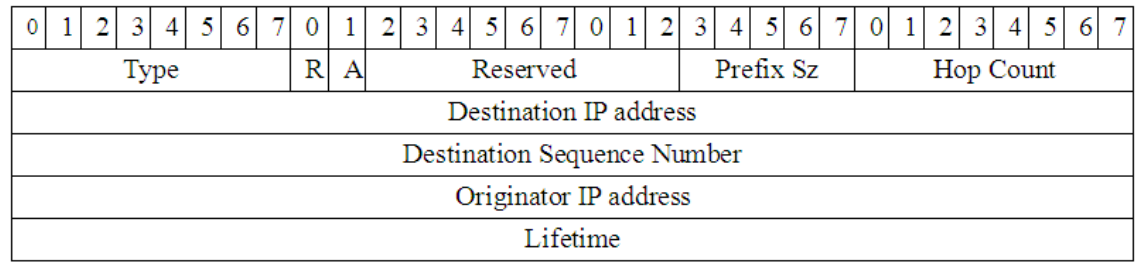
Struct RREP 如下截图所示：

aodv\_rrep.h



aodv\_rrep.h

在AODV协议中，RREP帧用于路由应答。其帧格式如下：



包含字段及其功能如表：

|  |  |
| --- | --- |
| 字段 | 功能 |
| Type | 2 |
| R | Repair flag修复标志；用于多播 |
| A | Acknowledgment required需要确认 |
| Resserved | 填充 0；接收时忽略此字段。 |
| Prefix Sz | 前缀长度。这个字段为5个bit（值为0到31），如果非0，则代表下一跳节点可以作为任何具有相同路由前缀的节点被请求时的目的节点。这个“相同路由前缀”就是Prefix Size定义的前缀。 |
| Hop Count | 从发起节点到目标节点的跳数。对多播路由请求，这个跳数则是从发起节点到多播节点组里产生 RREP信息的节点的跳数。 |
| Destination IP Address | 目标节点的IP地址，一条路由将提供给这个节点。 |
| Destination Sequence Number | 和这条路由联系在一起的目标序列号。 |
| Originator IP Address | 发起 RREQ 消息的节点的 IP 地址，路由将被提供给这个节点。 |
| Lifetime | 路由生命时间，单位为毫秒。在这段时间内，接收RREP的节点会认为这条路由是有效的。 |

注意到 Prefix Size（前缀长度）使得一个子网的路由器能够为子网内所有的主机提供路由信息，而这个所谓的“子网”则是由路由前缀来定义的。路由前缀由子网路由器和前缀长度共同决定。为了让这个特性发挥作用，子网路由器必须保证它和其他所有具有相同子网前缀的主机都是可达的。当前缀长度非零时，所有的路由信息（包括先驱信息）都必须遵循子网路由，而不是单独的包含子网内目的主机的地址。

### 3.6.2 RREP 相关函数

在aodv\_rrep.c，aodv\_rrep.h文件中，提供了如下表所示的函数，用于对rrep帧进行操作。

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| RREP \*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); | 本函数根据传进的参数分别对新建的rrep消息初始化。 |
| RREP\_ack \*rrep\_ack\_create(); | rrep路由应答确认创建函数。 |
| AODV\_ext \*rrep\_add\_ext  (RREP \* rrep, int type, unsigned int offset, int len, char \*data); | rrep扩展信息添加函数。 |
| void rrep\_send  (RREP \* rrep, rt\_table\_t \* rev\_rt, rt\_table\_t \* fwd\_rt, int size); | rrep消息发送函数。 |
| void rrep\_forward  (RREP \* rrep, int size, rt\_table\_t \* rev\_rt, rt\_table\_t \* fwd\_rt, int ttl); | rrep消息转发函数。 |
| void rrep\_process  (RREP \* rrep, int rreplen, struct in\_addr ip\_src, struct in\_addr ip\_dst, int ip\_ttl, unsigned int ifindex); | rrep消息处理函数 |
| void rrep\_ack\_process  (RREP\_ack \* rrep\_ack, int rreplen, struct in\_addr ip\_src, struct in\_addr ip\_dst); | rrep\_ack消息处理函数。 |

### 3.6.2 RREP 相关知识

当 RREQ 最终到达目的节点时，目的节点通过向该反向路由（即该RREQ传播路线）发送 RREP 应答帧，从而在该条路径的各个节点建立通向目的节点的前向路由。

只有在以下情况下节点才会产生 RREP:

（1） 该节点本身就是目的节点；

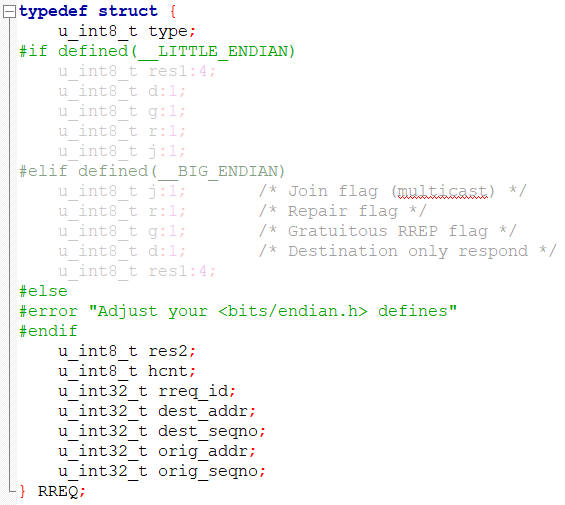
（2） 该节点为中间节点，但是它有通向目的节点的活跃路径；

当RREP传播到源节点时，中间节点根据该RREP更新它们各自指向目的节点的路由信息。同时节点只对第一次收到的RREQ发送RREP应答帧，之后到达的 RREQ 将被忽略。

## struct RREQ

### 3.7.1 RREQ 结构

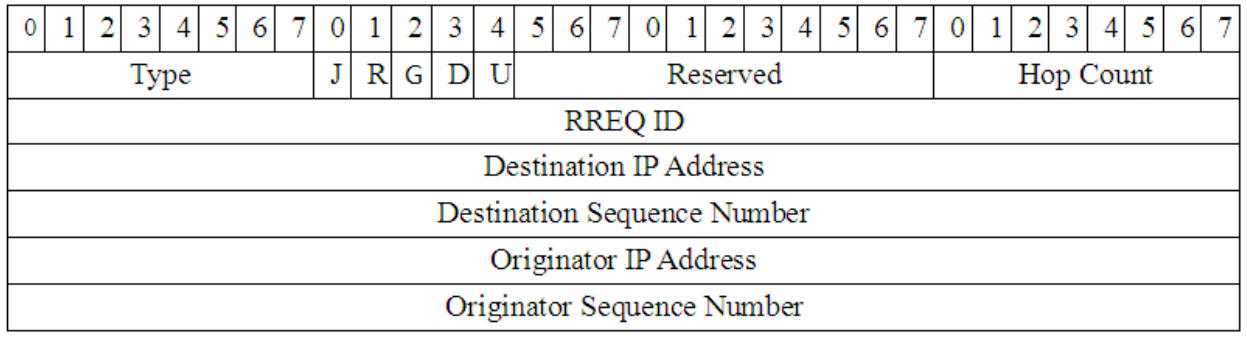
aodv\_rreq.h



aodv\_rreq.h

图3.3 struct RREQ的代码定义

在AODV协议中，RREQ帧用于路由请求。其帧格式如下：



各字段及其功能如下表：

|  |  |
| --- | --- |
| 字段 | 功能 |
| Type | 1 |
| J | Join flag（加入标志）：为多播保留。 |
| R | Repair flag（修复标志）：为多播保留。 |
| G | Gratuitous RREP flag（免费路由回复标志）： 指示是否该向目标节点 IP 地址域指定的节点发送一个免费路由回复消息。 |
| D | Destination only flag （仅允许目的节点回复标志）：标志置位则仅允许目的节点回复本条路由请求。 |
| U | Unknown sequence number（未知序列号）： 指示目标节点序列号未知。 |
| Reserved | Sent as 0（填充0）：接收端忽略此字段。 |
| Hop Count | 从发起节点到处理该请求的节点的跳数。 |
| RREQ ID | 这是一个序列号，用它和发起节点的IP就可以唯一标识一个RREQ信息。 |
| Dsetination IP Address | 目的节点的IP地址，本RREQ消息的任务就是想在发起节点和目的节点之间建立一条路由。 |
| Destination Sequence Number | 发起节点在以前通往目标节点的路由信息中能找到的最新的序列号。 |
| Originator IP Address | 发起本条路由请求消息的节点的IP地址。 |
| Originator Sequence Number | 指向发起者的路由表项中正在使用的序列号。 |

### 3.7.2 RREQ 相关函数

在aodv\_rreq.c，aodv\_rreq.h文件中，提供了如下表所示的函数，用于对rrep帧进行操作。

|  |  |
| --- | --- |
| RREQ \*rreq\_create  (u\_int8\_t flags, struct in\_addr dest\_addr, u\_int32\_t dest\_seqno, struct in\_addr orig\_addr); | rreq创建函数。 |
| void rreq\_send (struct in\_addr dest\_addr, u\_int32\_t dest\_seqno, int ttl, u\_int8\_t flags); | rreq消息发送函数。 |
| void rreq\_forward(RREQ \* rreq, int size, int ttl); | rreq消息转发函数。 |
| void rreq\_process(RREQ \* rreq, int rreqlen, struct in\_addr ip\_src, struct in\_addr ip\_dst, int ip\_ttl, unsigned int ifindex); | 接收到的rreq消息的处理函数。 |
| void rreq\_route\_discovery(struct in\_addr dest\_addr, u\_int8\_t flags, struct ip\_data \*ipd); | 为单播目的地执行路由发现的函数。 |
| void rreq\_record\_timeout(void \*arg); | rreq消息记录超时函数。 |
| struct blacklist \*rreq\_blacklist\_insert(struct in\_addr dest\_addr); | rreq黑名单插入函数。 |
| void rreq\_blacklist\_timeout(void \*arg); | rreq黑名单超时函数。 |
| void rreq\_local\_repair(rt\_table\_t \* rt, struct in\_addr src\_addr, struct ip\_data \*ipd); | 本地修复函数。 |

### 3.7.3 RREQ 相关知识

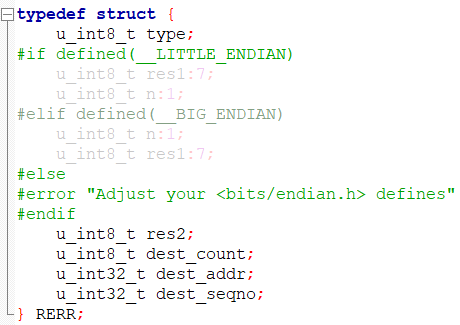
在两个节点之间的路由有效、通信正常的情况下，AODV路由协议不起任何作用。只有当源节点S需要向目的节点D发送数据包，但又没有D节点的路由入口时才会发起路由请求，即发送路由广播帧RREQ。 RREQ有以下要点：

* 当RREQ在网络中传播时，中间节点会更新各自到源节点的路由，我们称此路由为反向路由。
* RREQ请求帧中包含源节点以前记录的到目的节点的序列号，但此序列号可能不是最新的（最大的）。
* 中间节点如果有到目的节点的路由时，只有该节点记录的目的节点序列号比RREQ中的目的节点序列号更新（更大）时，才认为这条路由是有效的。

## Struct RRER

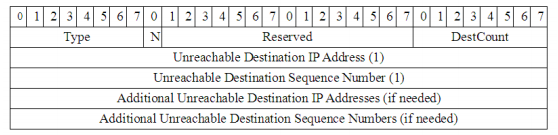
3.8.1 RRER 结构

aodv\_rrer.h



aodv\_rrer.h

在AODV协议中，RRER帧用于路由错误帧。其帧格式如下：



|  |  |
| --- | --- |
| 字段 | 功能 |
| Type | 3 |
| N | 不必删除标志；当一个节点已经对这条连接作了本地修复时，这个标志位置位，这样上游的节点就不用删除这条路由。 |
| Reserved | 保留字段，填充0，接收端不作处理。 |
| DestCount | 本消息内包含的不可达目的节点的数目，必须至少为1。 |
| Unreachable Destination IP Address | 因为连接断开而不可达的目的节点的IP地址。 |
| Unreachable Destination Sequence Number | 路由表项里不可达目的节点的序列号。 |

3.8.2 RRER 相关函数

在aodv\_rrer.c，aodv\_rrer.h文件中，提供了如下表所示的函数，用于对rrer帧进行操作。

|  |  |
| --- | --- |
| 函数原型 | 功能 |
| RERR \*rerr\_create  (u\_int8\_t flags, struct in\_addr dest\_addr, u\_int32\_t dest\_seqno); | rrer消息的创建函数。 |
| void rerr\_add\_udest  (RERR \* rerr, struct in\_addr udest, u\_int32\_t udest\_seqno); | 不可达目的节点添加函数。 |
| void rerr\_process  (RERR \* rerr, int rerrlen, struct in\_addr ip\_src, struct in\_addr ip\_dst); | rerr消息处理函数。 |

3.8.3 RRER 相关知识

发生以下情况，则广播RERR路由错误帧：

* 一个节点检测到与一个邻居节点的链路断裂（即该邻居节点不可达）；
* 节点收到收到一个数据包，而该节点路由表没有志向数据包制定的目的地址的有效路由，并且该路由并非处于修复状态。
* 节点收到来自邻居节点的RERR路由错误信息帧，该帧可能指示多个目的节点不可达。

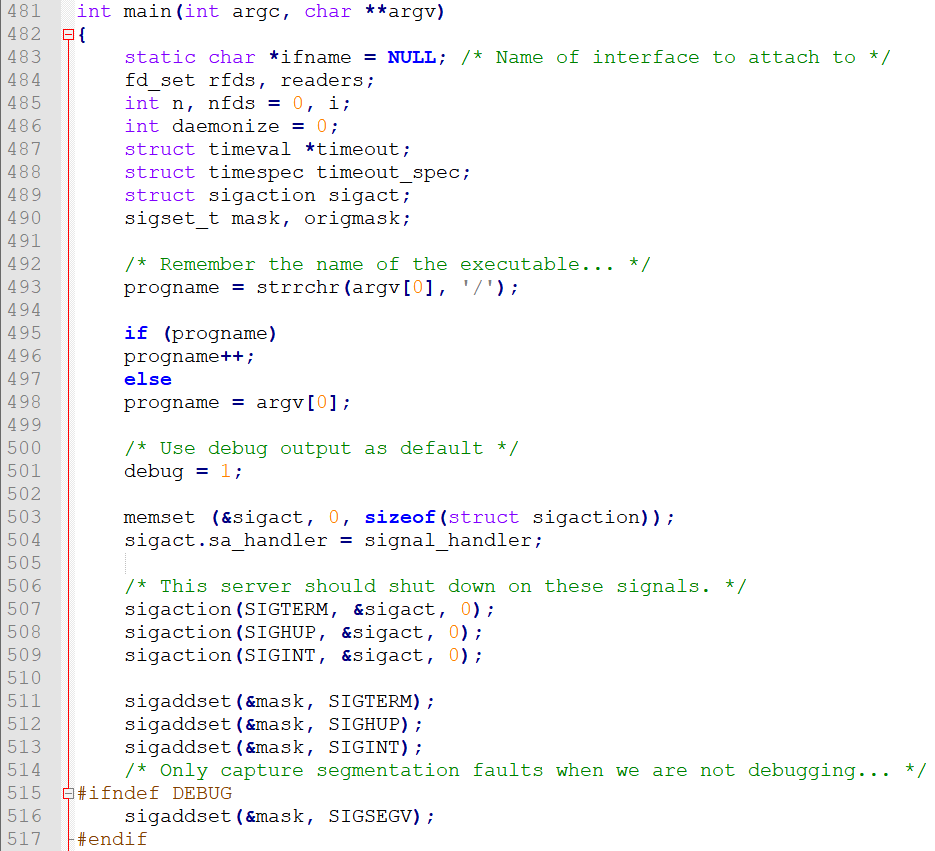
RERR信息的发送方式：单播（将RERR信息单播发送给一个接收者）、重复单播（将RERR信息分别的单播发送给多个接收者）、广播（将RERR信息同时发送给多个接收者，使用IP地址255.255.255.255进行广播，TTL=1）。

# 第四章 AODV 路由协议的实现

## main.c

### 4.1.1 main函数

main.c



main.c

484：定义了名为rfds，readers的fd\_set类型变量。

485：定义了相关变量。

486：int daemonize（守护进程）=0。

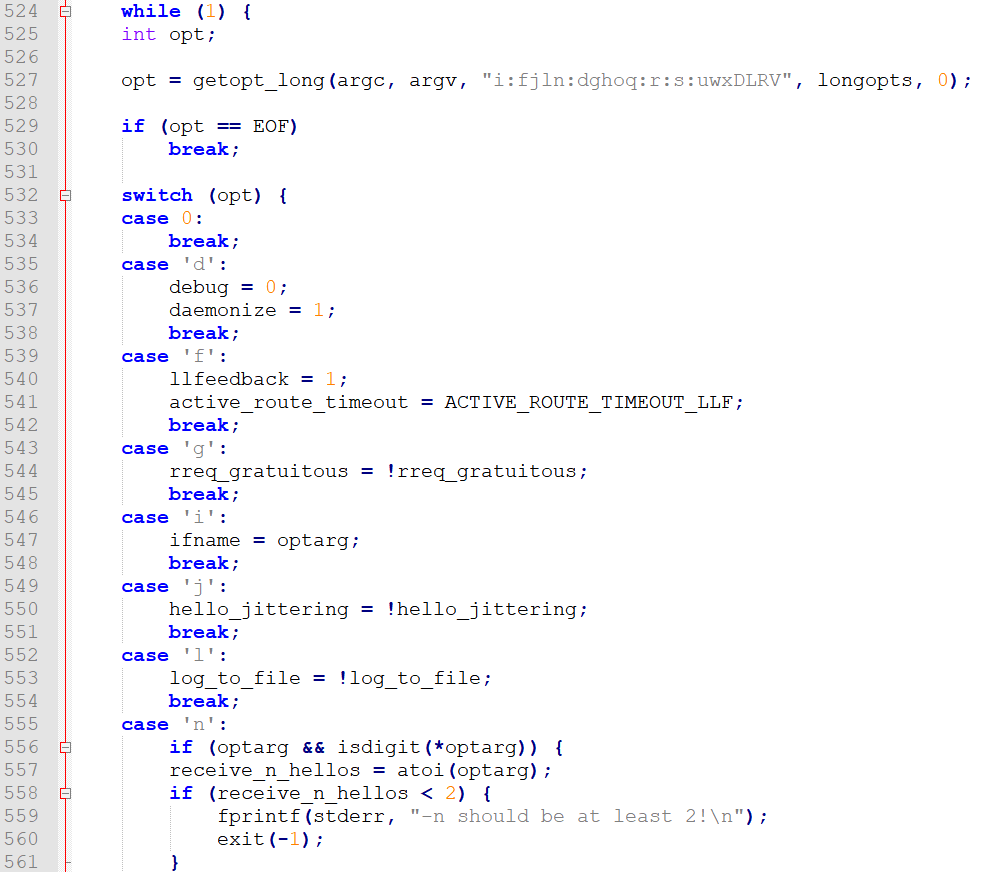
487：定义了timeval类型的变量指针 timeout。

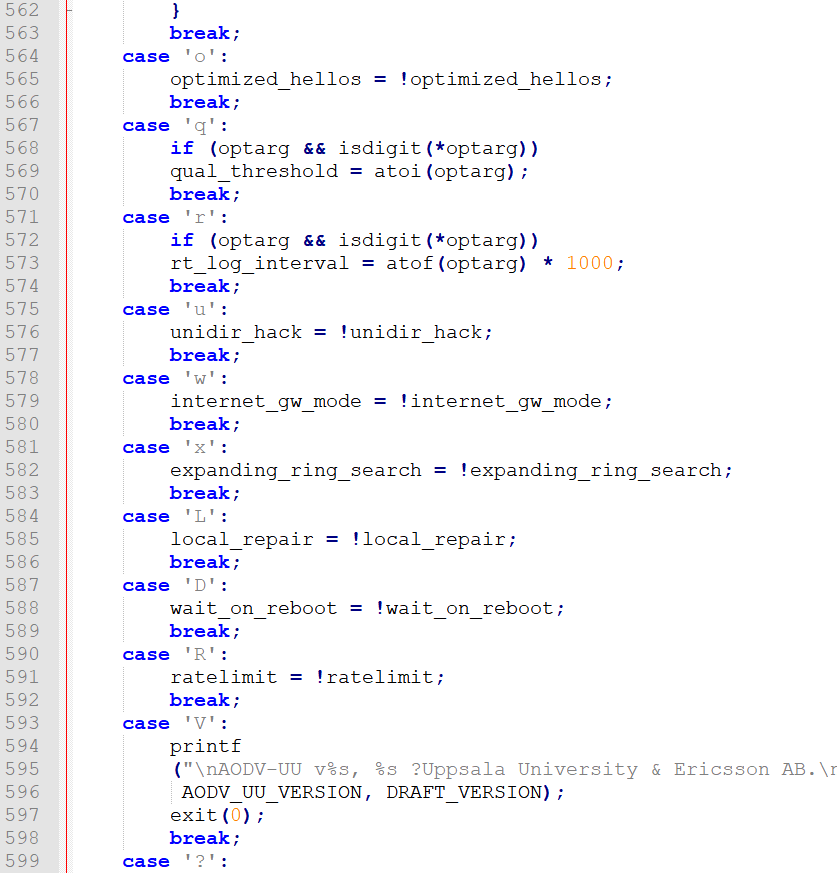
488：timespec变量timeout\_spec。

489：sigaction变量sigact。

493~513：进行一些准备工作，如获取程序名、初始化信号。

main.c



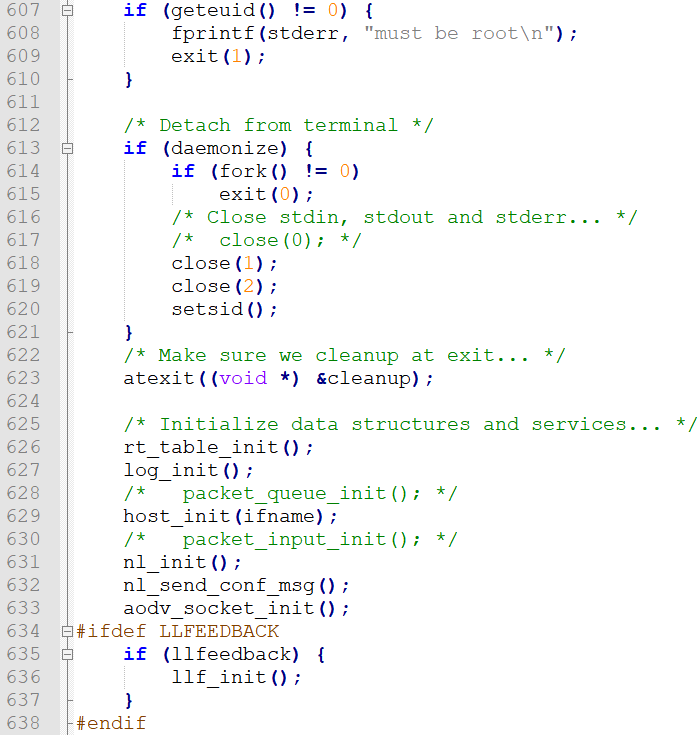


main.c

524~605：进入永真循环，根据读取的参数不同，执行不同的分支。具体情况如下：

|  |  |  |
| --- | --- | --- |
| 命令行参数 | 全称 | 功能 |
| -d | daemon | 开启守护进程模式，脱离控制台窗口。 |
| -g | force-gratuitous | 在每个RREQ消息上强制设置 gratuitous 标记。 |
| -h | help | 打印所有参数及含义。 |
| -i | interface | 表示要绑定的接口。 |
| -j | hello-jitter | 触发 hello-jitter 功能，默认情况下开启。 |
| -l | log | 输出日志。 |
| -o | opt-hellos | 设置只在转发数据包时发送 HELLO 消息。 |
| -r | log-rt-table | 每隔一段时间记录路由表。 |
| -n | n-hellos | 设置接到 n 个 HELLO 消息才当成邻居。 |
| -u | unidir-hack | 侦测并避免单向链路。 |
| -w | gateway-mode | 开启试验性的因特网网关支持。 |
| -x | no-expanding-ring | 禁用 RREQ 消息的扩展环搜索法。 |
| -D | no-worb | 禁用重启延迟的等待。 |
| -L | local-repair | 开启本地修复。 |
| -f | llfeedback | 开启链路层反馈。 |
| -R | rate-limit | 开启 RREQ 和 RERR 消息的速率限制。 |
| -q | quality-threshold | 为控制包设置一个信号质量最小阈值。 |
| -V | version | 输出版本信息。 |
| 其他 |  | 打印usage清单。 |

main.c



main.c

606~610：检查是否在root权限下执行.

613~621：检查daemonize变量值，为1进入if代码块。

614~615：开辟子进程，关闭父进程。

618~619：子进程关闭标准输入输出错误流。

620：调用setsid函数使子进程成为新的会话的领头进程，并与其父进程的会话组和进程组脱离。

623：调用atexit函数登记了函数cleanup，用于在main函数执行结束后，执行atexit中注册的函数cleanup()，用于在结束程序后的清理工作。

626~637：调用各初始化函数，进行数据接口，服务的初始化

626：初始化路由表。

627：初始化日志。

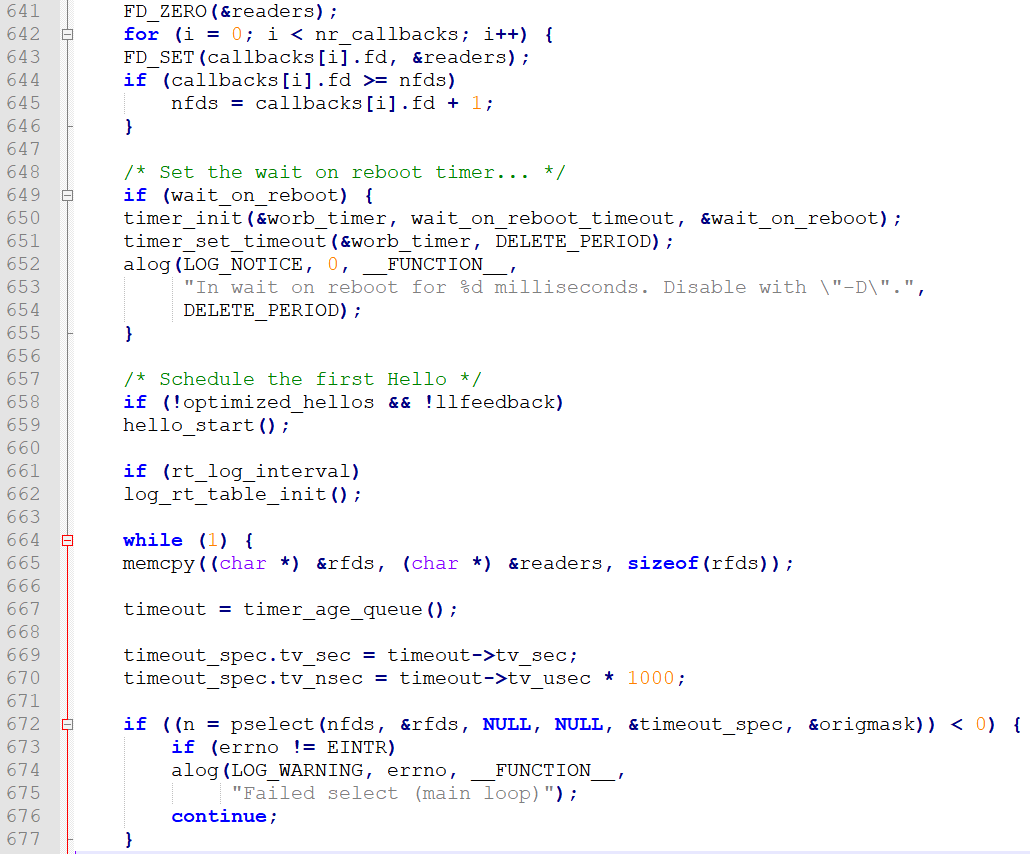
629：初始化host。

631~632：初始化nl。

633：初始化aodv\_socket。

635~636：若开启链路层反馈，则初始化链路层。

main.c



main.c

641~646：设置套接字。

641：清空文件描述符集readers，即初始化。

642~646：将回调数组中的每一个fd加入描述符集。

649~655：在重新启动计时器上设置等待，若超出一定时间，则重启。

649：if(wait\_on\_reboot)即未禁用重启延迟的等待，没有设置-D

650：初始化timer worb\_timer，初始化等待重启计时器

651：为worb\_timer增加DELETE\_PERIOD长度的时间

652~655：log记录

657~663：开始准备第一个hello帧（但不发送），初始化路由表。

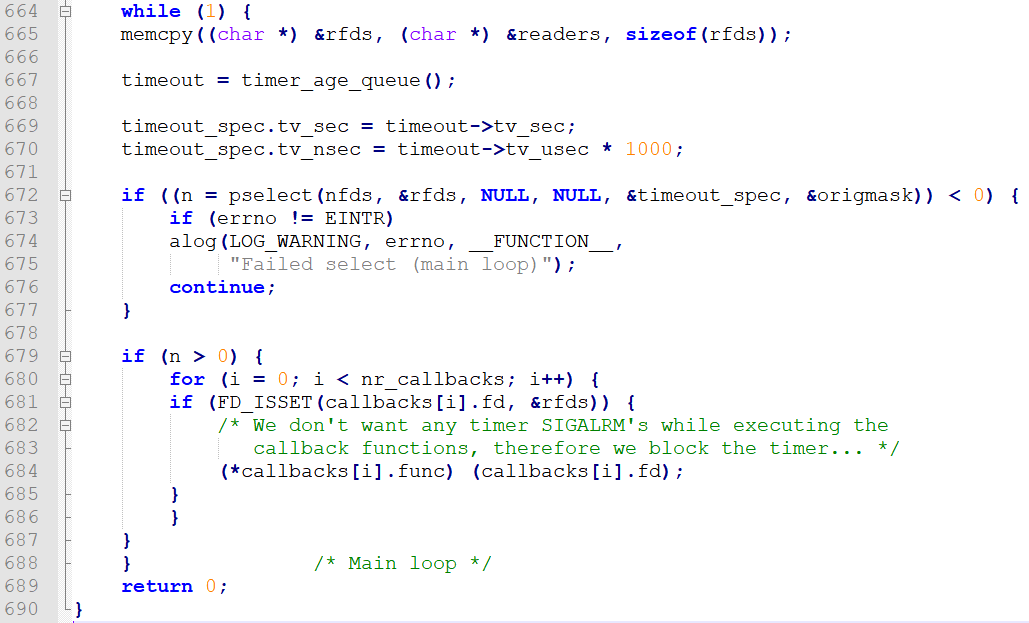
658：若不设置-o（只在转发数据包时发送hello）且不设置-f（开启链路层反馈）

659：调用发送hello消息的准备函数hello\_start()；（并未发送）

661:若设置-r,即每隔一段时间记录路由表

662：初始化路由表log\_rt\_table；

main.c



main.c

664~688:永真循环。

665：将readers的内容复制进rfds中

667：timeout = TQ队列中获取的新的timer

669~670：将timeout变得更加精确，至纳秒级，赋值给timeout\_spec

使用pselect函数进行IO复用，允许进程指示内核等待多个事件中任一个发生，并只在有一个或多个事件发生或经历一段指定的时间后才唤醒它，并调用相应的回调函数来处理。

672：调用pselect阻塞进程，当rfds可读时，唤醒进程

673~676：若返回值<0,即出错，则报错并进入下一次循环。

679~687：若n>0，即唤醒进程

680~686：在执行回调函数时，由于不需要任何定时器SIGALRM，故阻塞定时器。

684：函数指针执行回调函数，处理响应事件。

689：return 0；

### 4.1.2 main.c中其他函数

main.c中提供了如下表的相关函数。

|  |  |
| --- | --- |
| 函数原型 | 作用 |
| static void cleanup() | 清空套接字、路由表等数据结构和模块。 |
| void usage(int status) | 打印出所有命令行参数和其作用。 |
| int set\_kernel\_options() | 设置内核选项在所有接口上禁用ICMP重定向。 |
| int find\_default\_gw(void) | 寻找默认网关。 |
| Struct sockaddr\_in \*get\_if\_info  (char \*ifname, int type) | 根据名字和类型，寻找相应的接口，返回其信息。 |
| int attach\_callback\_func  (int fd, callback\_func\_t func) | 增加一个callback数据结构元素，包括一个描述符和一个函数。 |
| void load\_modules  ( char \*ifname) | 装载内核模块，如果模块在当前目录中则使用，否则调用 modprobe。 |
| void remove\_modules(void) | 删除某个模块。 |
| void host\_init(char \*ifname) | 初始化某个端口。 |
| void signal\_handler(int type) | 信号处理器，根据信号种类做出不同的处理。 |

## AODV路由发现

1. 广播RREQ路由请求帧。
2. 中间节点更新各自到源节点的路由表。
3. 如果收到RREQ的节点不是目的节点，并且没有到达目的节点的更新的有效路由，则转发该RREQ。
4. 中间节点维护指向路由源节点的反向路由。
5. 目的节点或存在到目的节点有效路由的中间节点产生RREP路由应答帧。
6. RREP通过之前建立的反向节点单播至源节点。
7. 源节点收到RREP应答帧，至此源节点可以向目的节点发送数据包。

### 4.2.1 广播RREQ路由请求帧

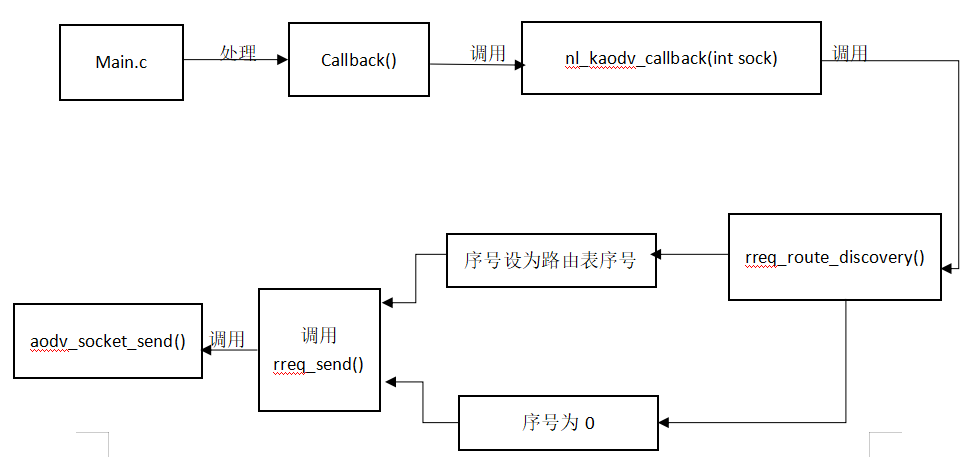
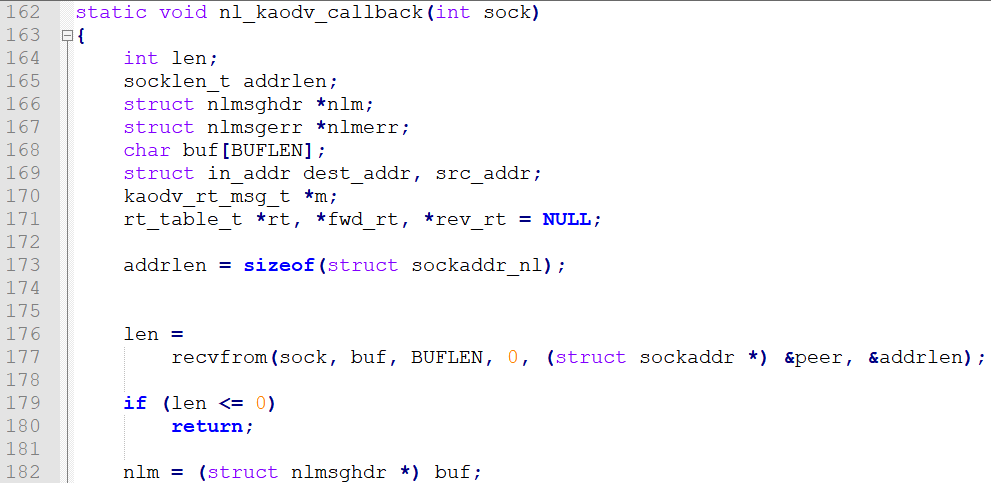
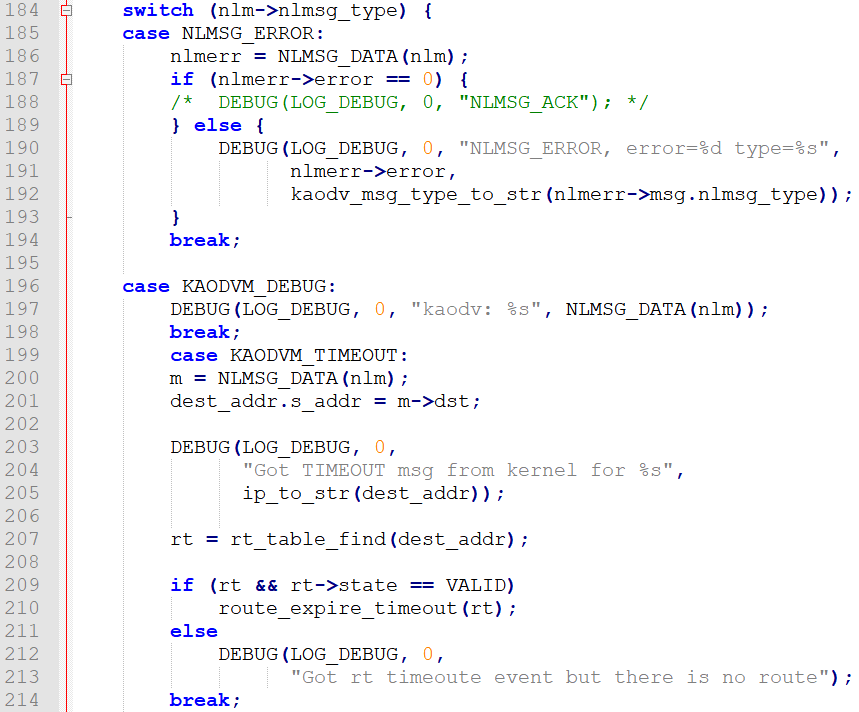


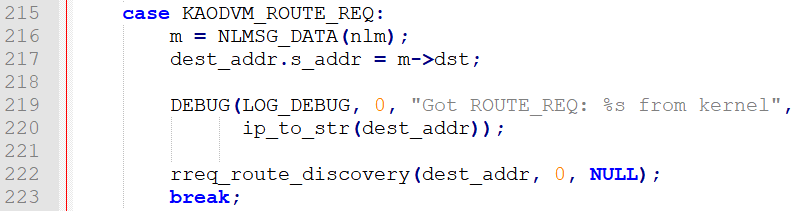
图4.1 节点处理RREQ消息的函数调用关系

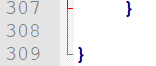
广播RREQ路由请求帧过程如图4.1所示。在main函数中，通过回调函数处理事件。当需要执行路由发现的时候，会调用函数nl\_kaodv\_callback(int sock);函数代码如下所示：

nl.c









nl.c

164~182：初始化各项变量。

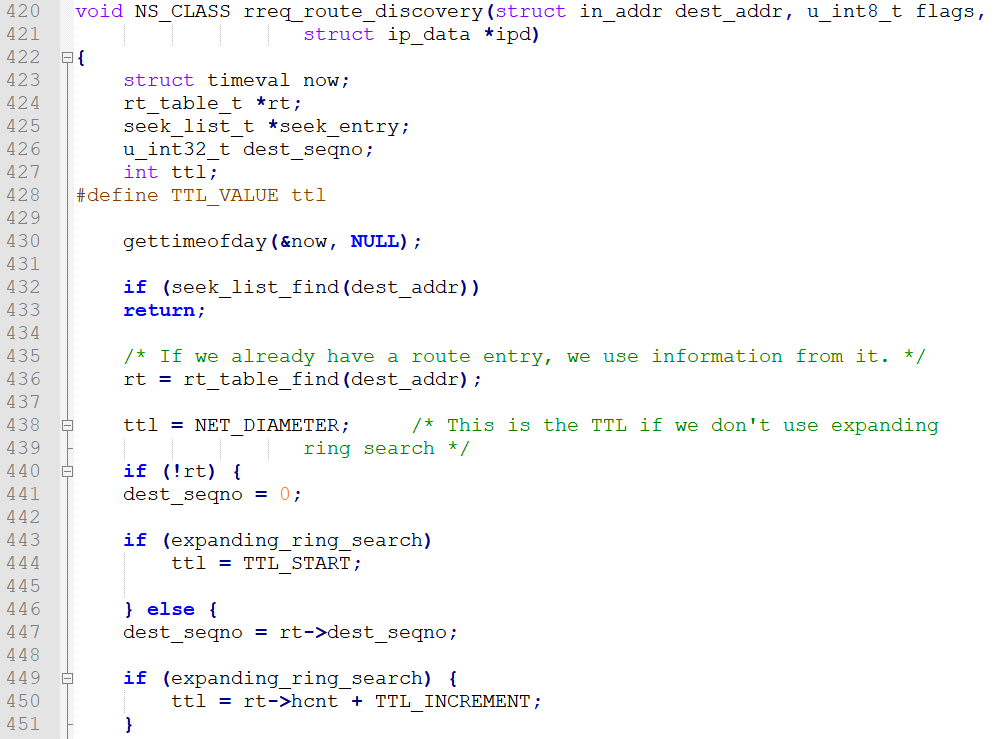
215~223：内核判断需要进行路由发现，便执行相关代码。

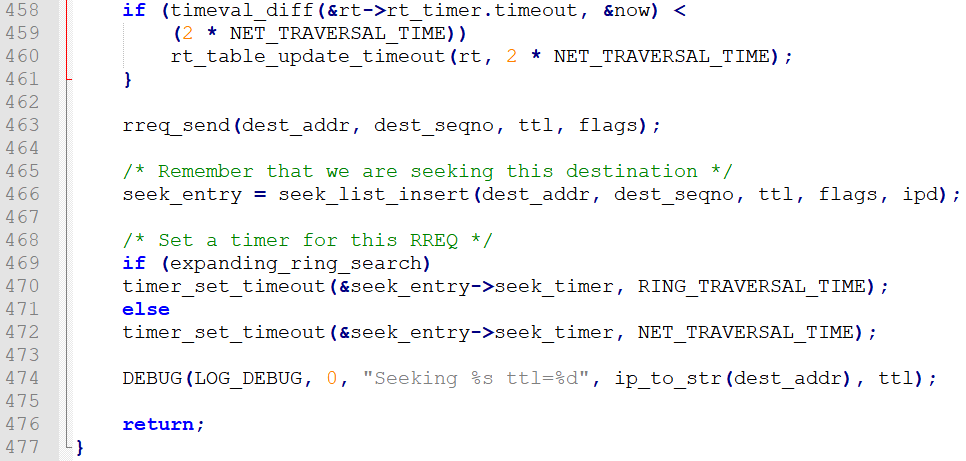
216~217：进行相关变量赋值。

219~220：debug相关操作。

222：调用rreq\_route\_discovery函数，为单播目的地发现路由。

aodv\_rreq.c





aodv\_rreq.c

420-477：为单播目的地执行路由发现的函数rreq\_route\_discovery函数。

436-438：根据目标地址，寻找路由表中是否有相应的表项。ttl的值设置为NET\_DIAMETER。

440-451：如果不使用扩展环搜索，这就是ttl。

440~446：如果没有找到路由表项，将变量dest\_seqno（rreq消息中的目标序号）路由表项的序号设置为0。如果禁用拓展环搜索法，则ttl=TTL\_START；

447~451：如果找到路由表项，将dest\_seqno设置为已查到的路由表项的序列号。如果禁用拓展环搜索法，则ttl=rt->hcnt + TTL\_INCREMENT；

470-472：为RREQ消息设置定时器。

458-461：等待RREP的路由表条目不应该在2\*NET\_TRAVERSAL\_TIME之前被清除

如果当前时间与定时器中时间差小于2\*NET\_TRAVERSAL\_TIME，则更新计时器，为计时器加上2\*NET\_TRAVERSAL\_TIME。

463：调用rreq\_send函数发送rreq消息。

466：记住这个我们正在寻找的目的地，将其加入seek\_list。

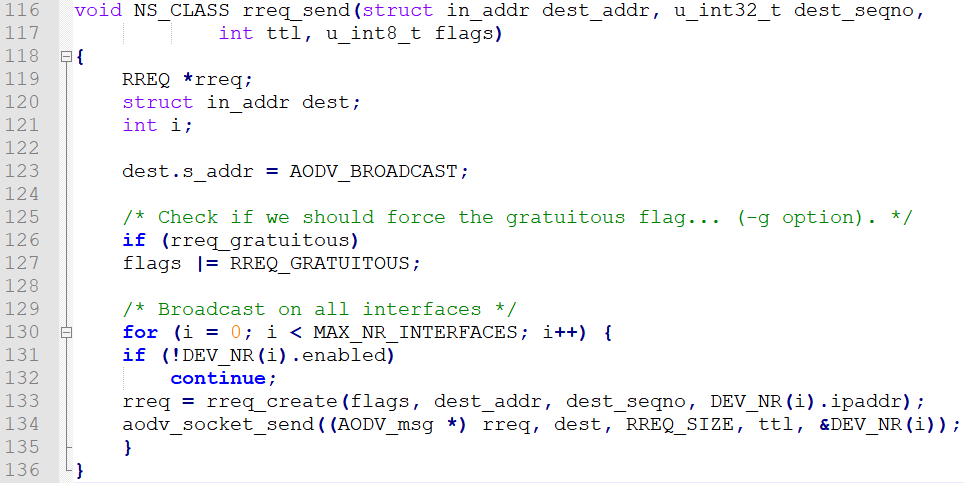
469~472：为这个rreq消息设置定时器。根据是否禁用拓展环算法，计时器值不同。

474：日志记录。

476：return；

rreq\_route\_discovery函数调用rreq\_send函数广播发送rreq消息。rreq\_send函数源码如下：

aodv\_rreq.c



aodv\_rreq.c

116-136：发送rreq消息的函数rreq\_send函数。

119-121：变量声明。

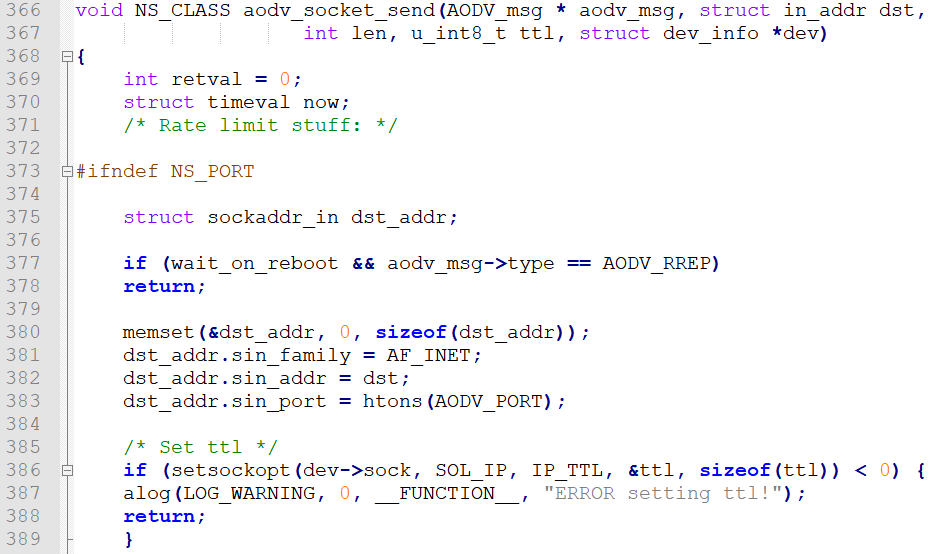
123：设置属性为广播。

126-127：检查我们是否应该强制执行无偿标志……（-G选项）。

130-135：在所有接口广播rreq消息，由rreq\_create()创建rreq消息，由aodv\_socket\_send函数发送消息。其中131-132 检查端口是否开启aodv路由。

rreq\_send函数调用了aodv\_socket\_send函数，通过套接字将rreq消息广播。aodv\_socket\_send函数源码如下：

aodv\_socket.c



aodv\_socket.c

369~375：声明变量；

377~378：边界检查。

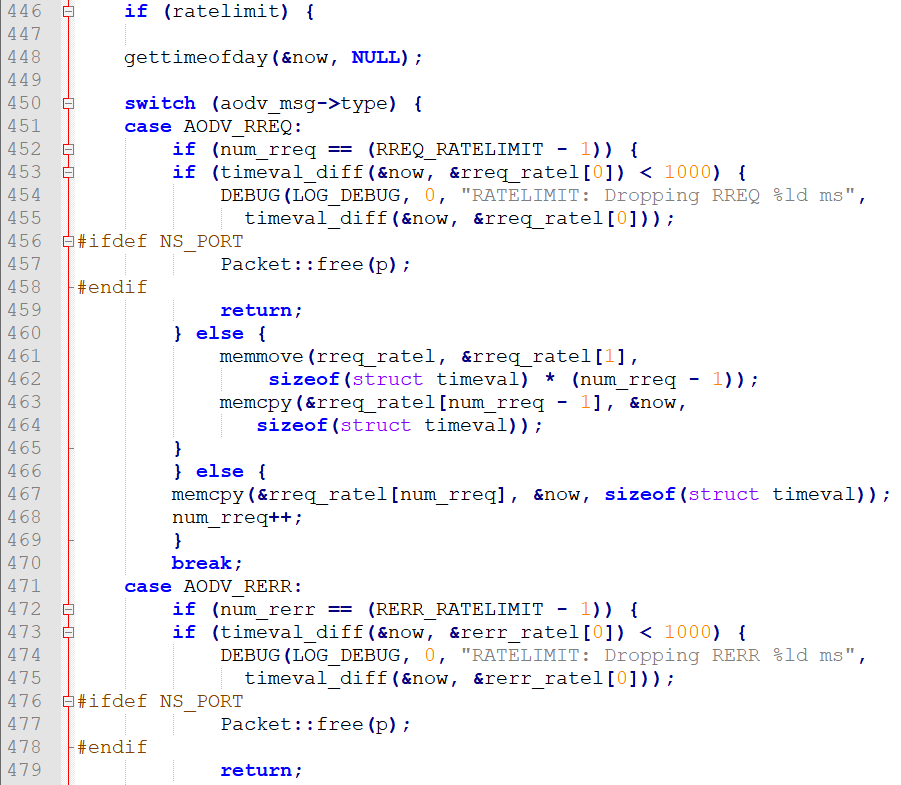
380~383：套接字相关设置。

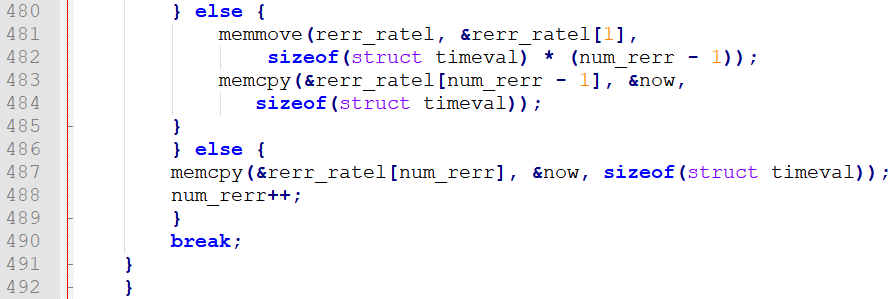
386~389：日志记录。

390~438：设置首部信息，设置 ttl，检查当前状态是否允许 RREP，清楚报文的AODV 部分，把要发送的消息复制到报文相应位置，设置其他常用首部信息，设置接收方的端口号。

由于篇幅限制，这里不贴代码。

aodv\_socket.c





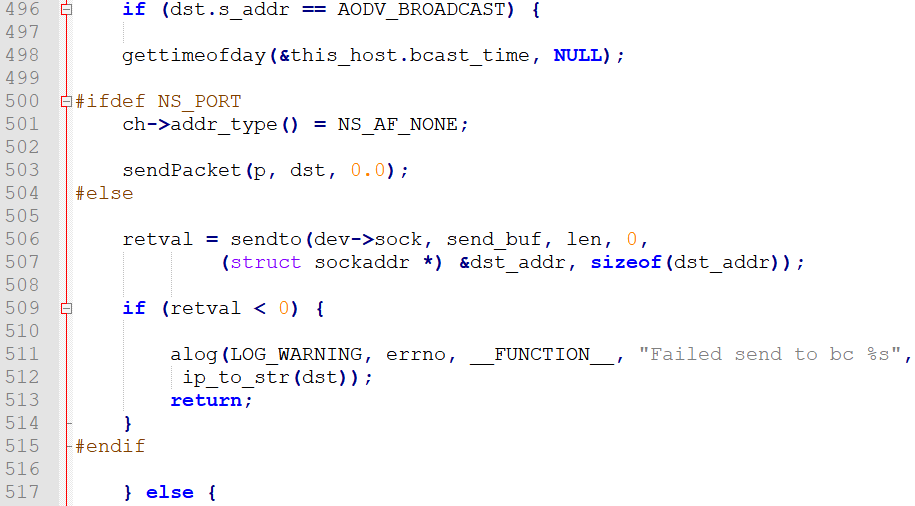
aodv\_socket.c

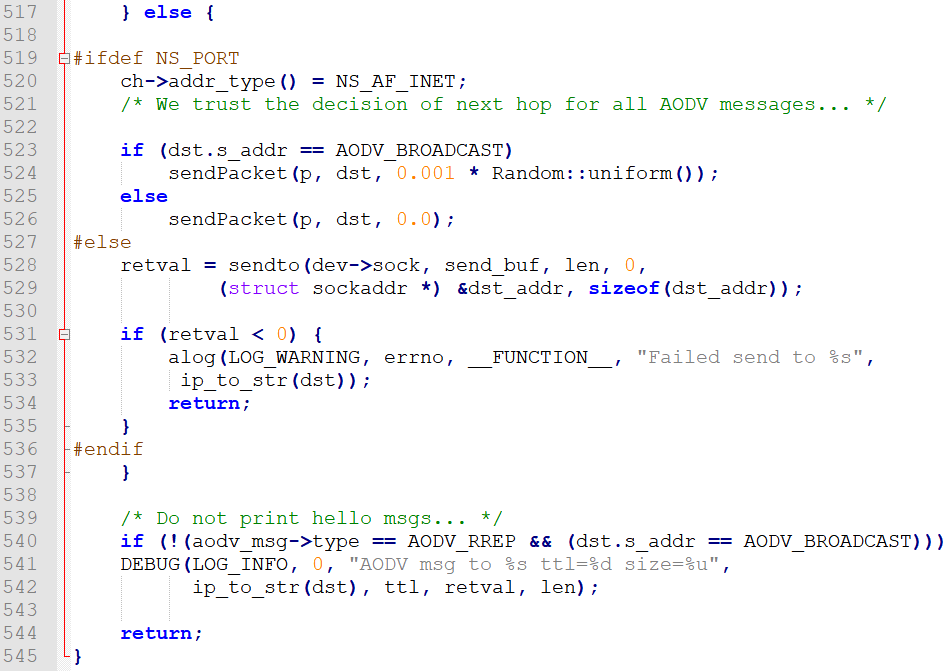
446~491：如果开启速率限制，检查我们将要发送的是RREQ还是RREP，如果这种数据包的最低传输速率比限定的值小，就丢弃该数据包。

450：判断要发送的消息类型

451~470：如果要发送rreq消息，则根据相关判断，决定是否丢弃该数据包。

aodv\_socket.c





aodv\_socket.c

496~535：根据需要发送这个数据包。

540~543：如果这是个广播数据，则更新最后广播该数据的时间以避免对HELLO消息的不必要广播。

由此完成了广播RREQ路由请求帧的工作。

### 4.2.2 中间节点更新各自到源节点的路由表

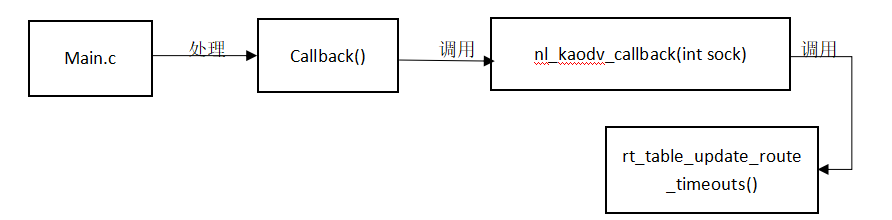
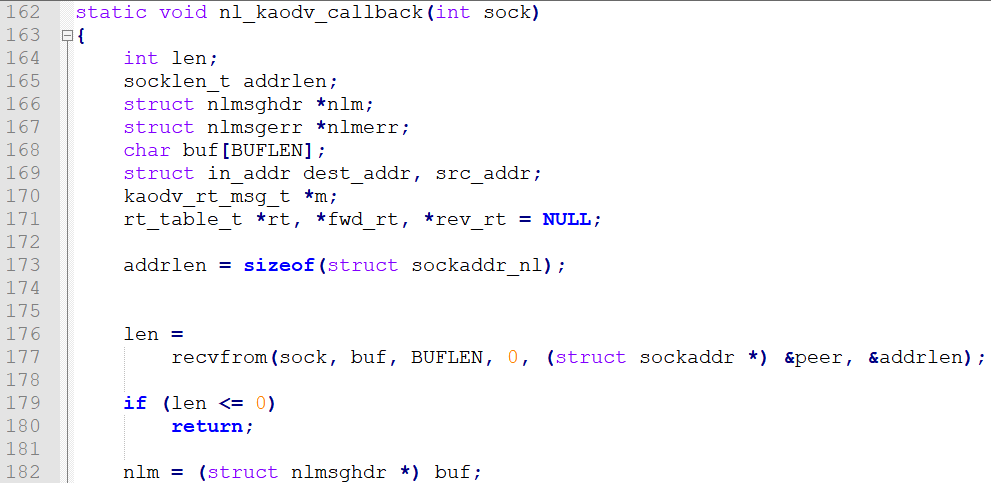
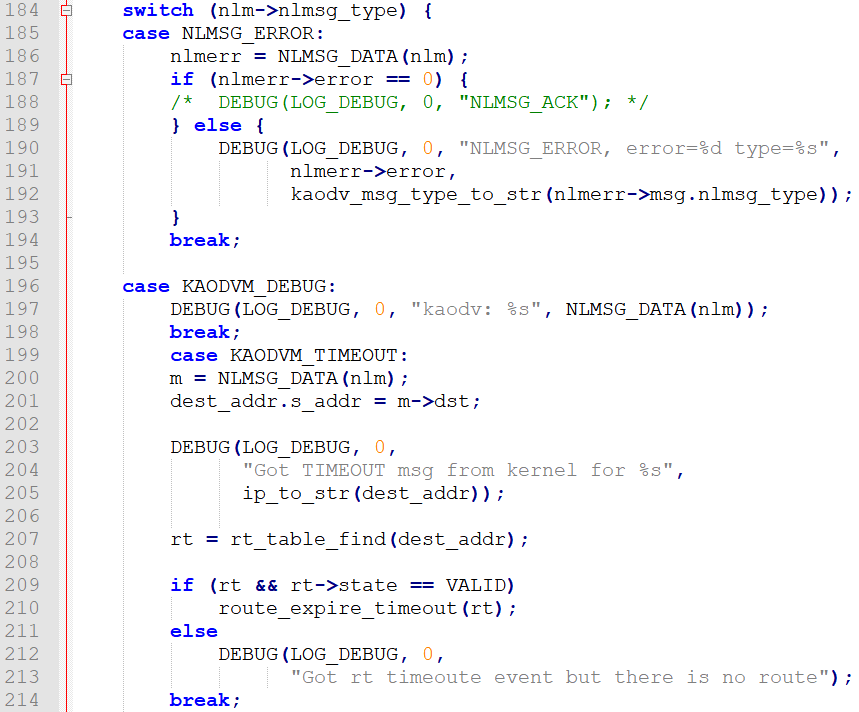


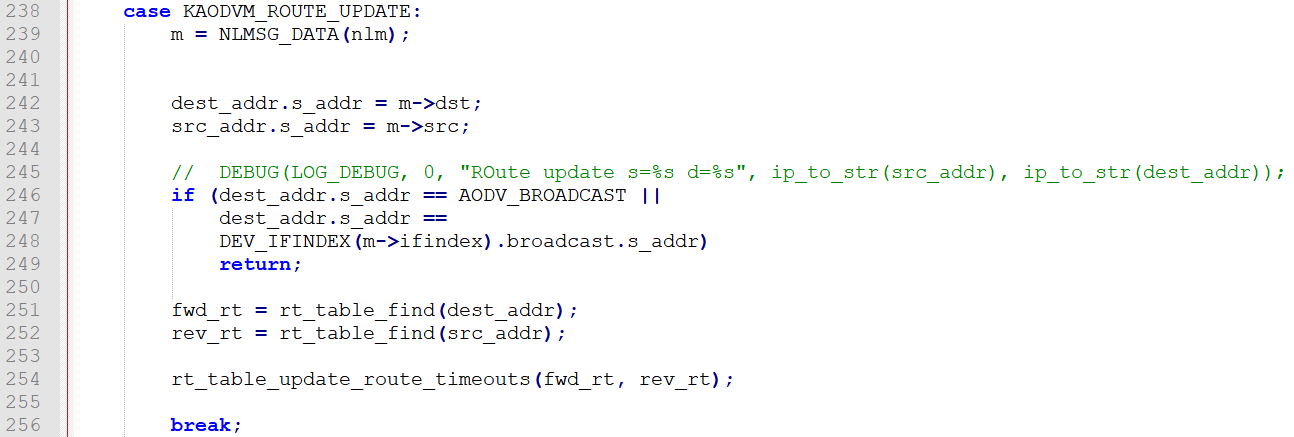
图4.2 中间节点更新路由表函数调用关系

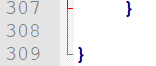
中间节点更新各自到源节点的路由表的过程如图4.2所示。在main函数中，通过回调函数处理事件。当需要执行中间节点更新路由表的时候，会调用函数nl\_kaodv\_callback(int sock);函数代码如下所示，由于前文已经描述过函数nl\_kaodv\_callback，故重复内容不再赘述，仅分析核心代码：

nl.c









nl.c

238~256：内核判断需要进行路由发现，便执行相关代码。

239~243：相关变量赋值。

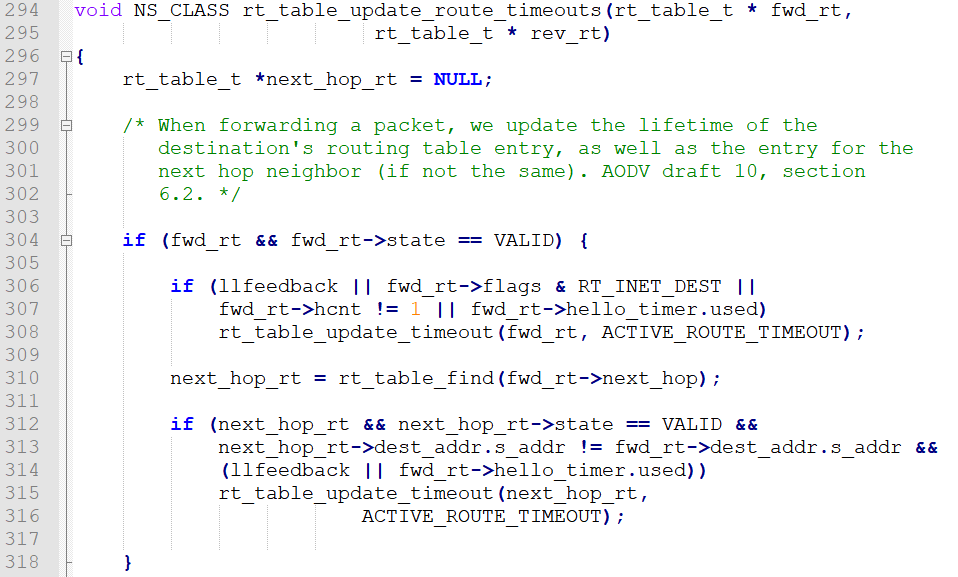
246~249：边界检查。

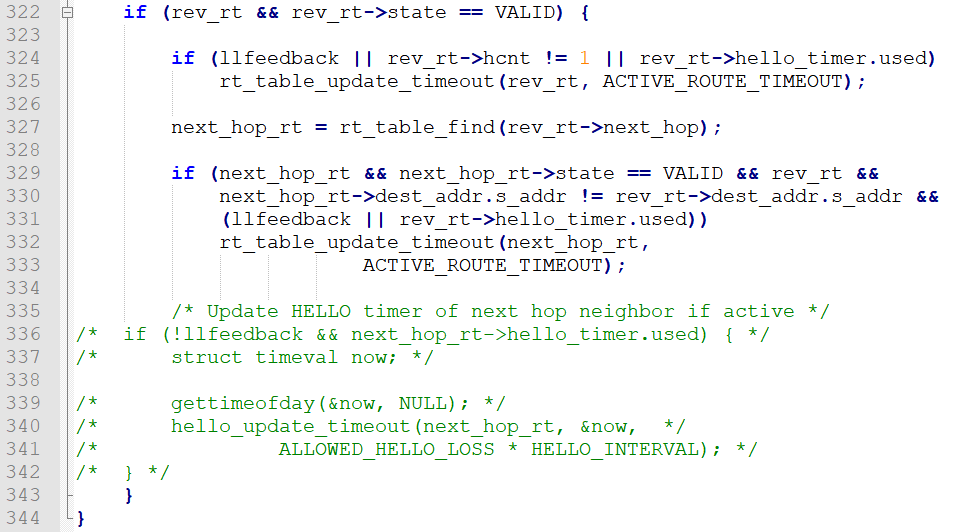
251~252：在路由表中查找地址为rreq消息源ip地址或rreq目的ip地址的路由表项。

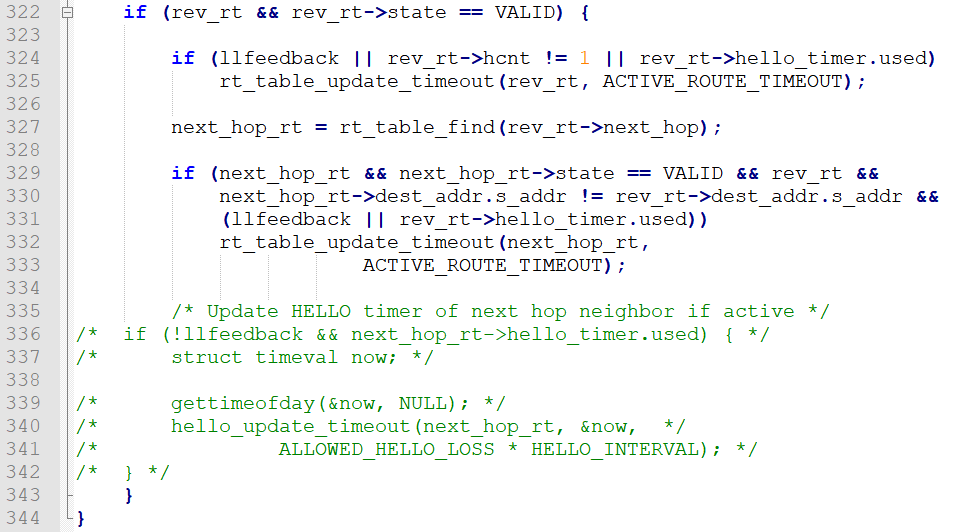
254：调用函数rt\_table\_update\_route\_timeouts，更新查找到的路由表项的计时器。

rt\_table\_update\_route\_timeouts函数源码如下：

routing\_table.c







routing\_table.c

294~344：函数rt\_table\_update\_route\_timeouts，功能：更新输入(rec\_vt)或输出(fwd\_rt)包路由时的定时器。

297：定义路由表项next\_hop\_rt(下一跳表项)；

在转发数据包时，我们更新目的地的路由表条目的生存期以及下一跳邻居的条目（如果不相同）

304：if(转发表项存在&&转发表项有效)

306：如果开启链路层反馈或转发表项的hello.timer已经被使用或满足其他条件

308：则更新fwd\_rt中计时器，生命周期为ACTIVE\_ROUTE\_TIMEOUT；

310：下一跳next\_hop\_rt = fwd\_rt的下一跳路由表项

312：如果下一跳表项存在&&下一跳状态有效&&下一跳目的IP地址等于fwd\_rt的目的IP地址&&(开启链路层反馈或转发表项的hello.timer已经被使用)

315：则更新下一跳表项的计时器，生命周期为ACTIVE\_ROUTE\_TIMEOUT；

因为发起者和目的地之间的路由应该是对称的，故沿着反向路径反向下一跳更新反向路由。

322~343：沿反向路径反向下一跳更新反向路由，基本同上，不再赘述。

### 4.2.3 中间节点处理RREQ消息

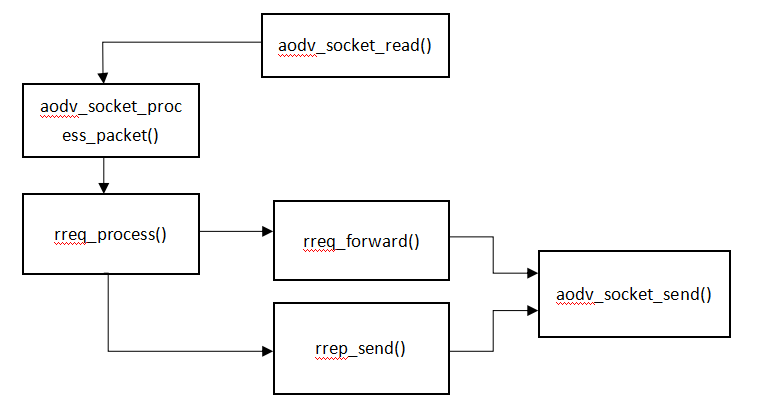
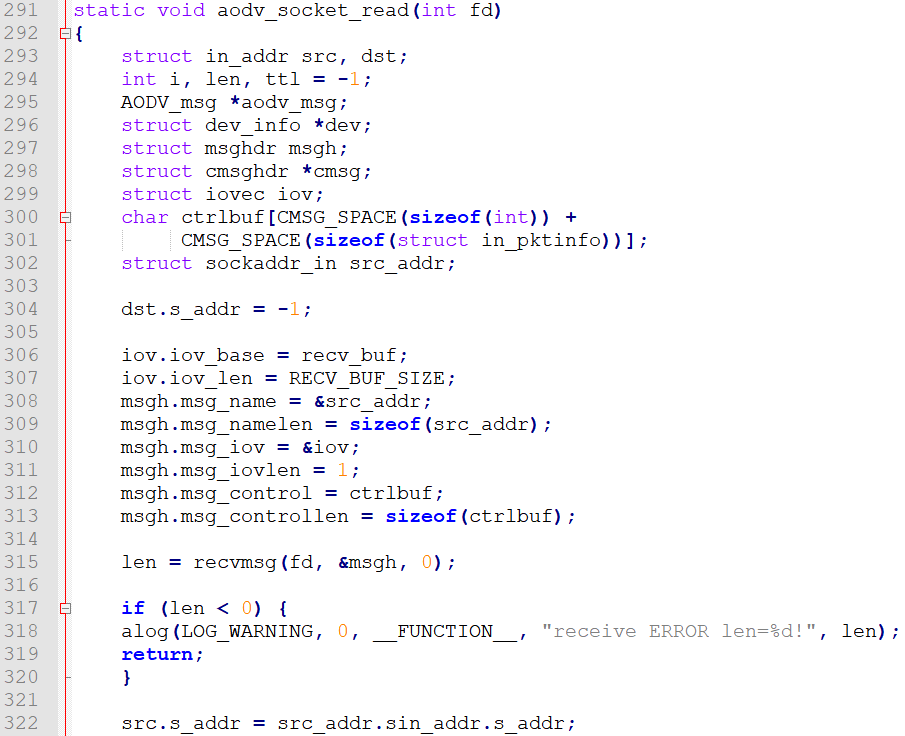
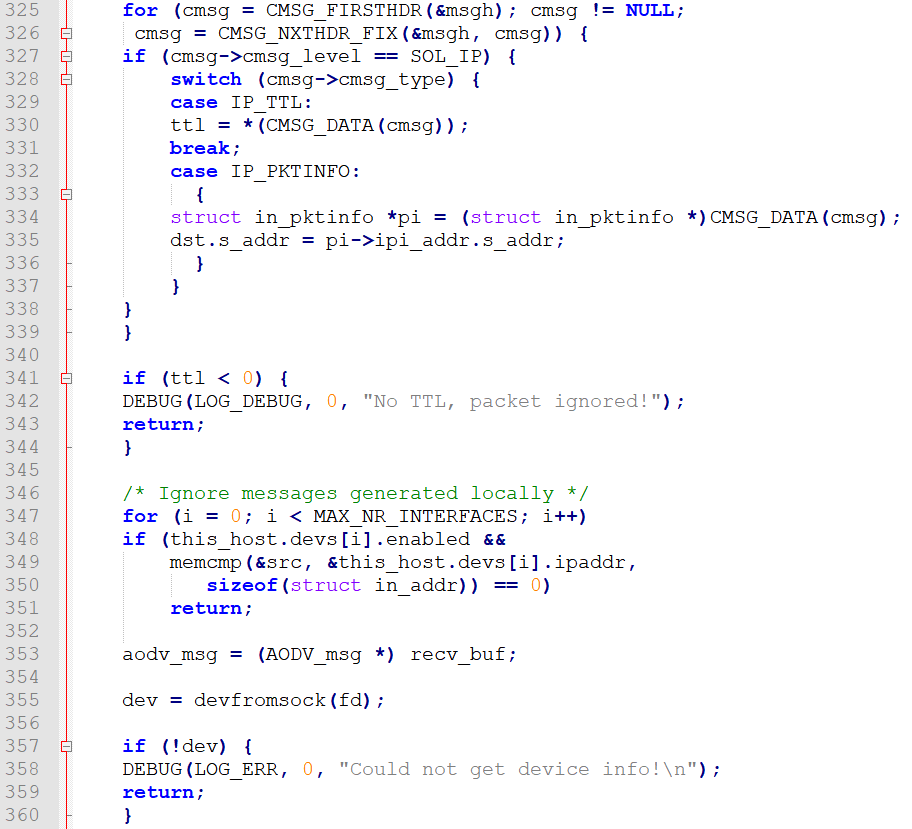


图4.3 中间节点收到RREQ消息时的处理函数调用关系

如果收到RREQ的节点不是目的节点，即中间节点。并且没有到达目的节点的更新的有效路由，则转发该RREQ。同时也会维护指向路由源节点的反向路由。其处理过程如图4.3所示。易知通过回调函数，调用了aodv\_socket\_read函数。其源码如下：

aodv\_socket.c







aodv\_socket.c

293~322：从套接字中获取相关信息。

325-339：从控制消息中获取ttl和目的地地址。

341-344：如果ttl<0,则忽略此数据包。

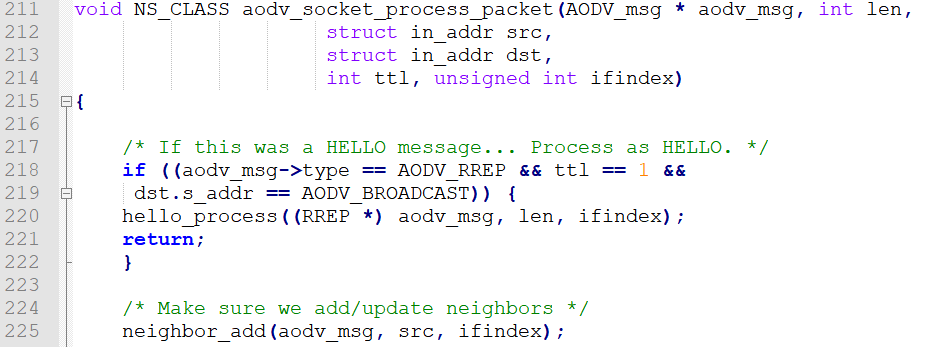
347-351：如果是本地产生的数据包，就忽略掉。

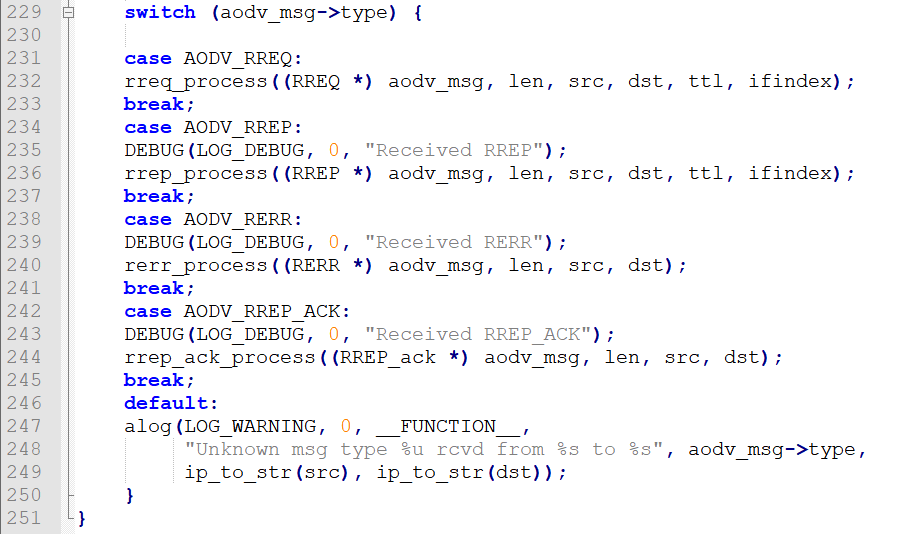
355-359：获取套接字中的device info。

362：调用aodv\_socket\_process\_packet函数对消息进行分类和处理。

aodv\_socket\_process\_packet函数源码如下：

aodv\_socket.c





aodv\_socket.c

211-251：aodv\_socket\_process\_packet函数的声明，此函数负责辨别接收到的包的类型，并且对不同类型的包进行不同处理，此段代码负责此函数的实现。

218-222：如果包是一个HELLO消息，就把控制传递给hello\_process函数。

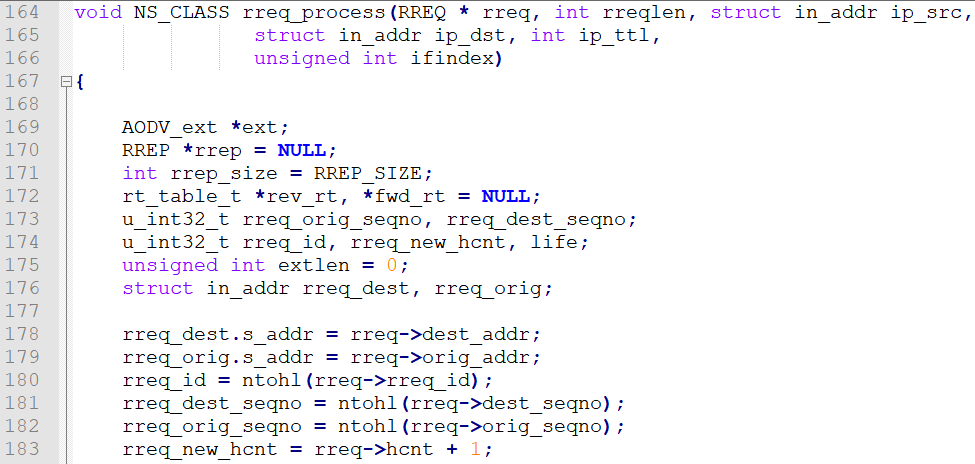
225：根据消息所包含的信息添加或更新邻居节点的信息。

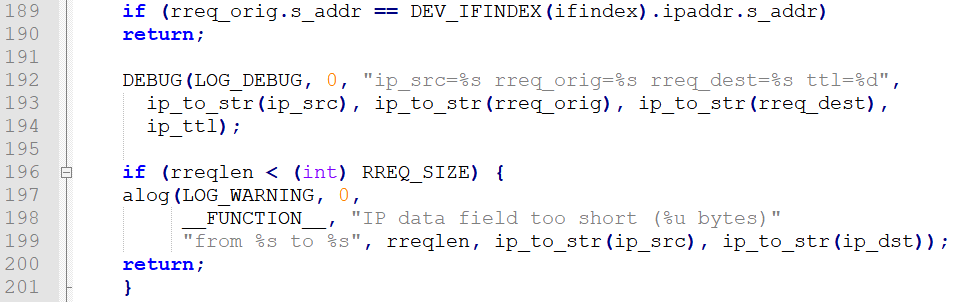
229-250：如果是HELLO以外的消息，根据类型记录路由日志并处理收到的消息。

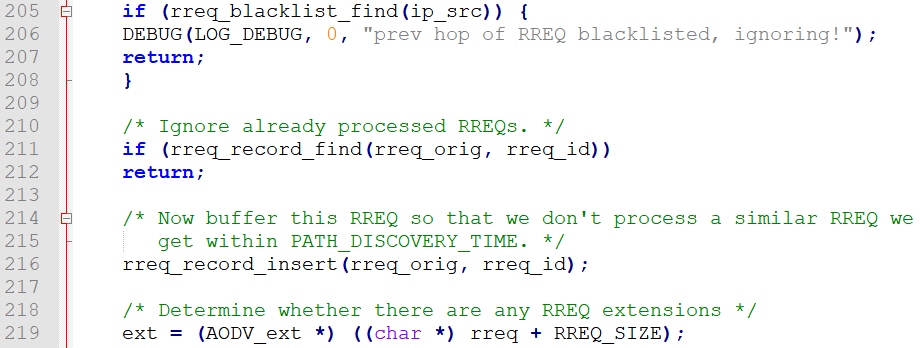
在231~233行中，该函数调用了rreq\_process函数，处理rreq消息。

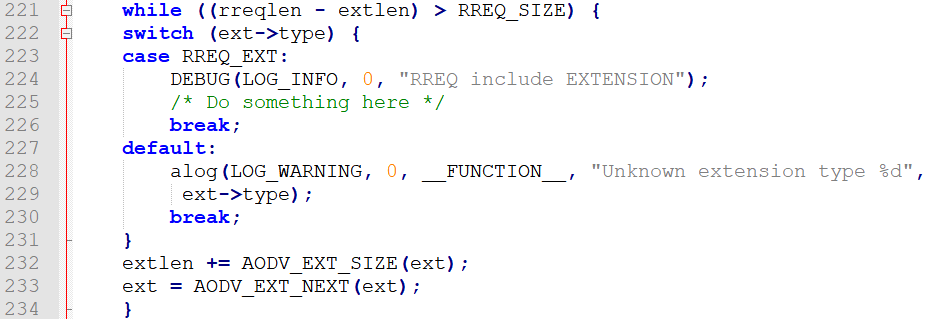
rreq\_process函数源码如下，由于该函数篇幅过长，这里仅截取核心部分：

aodv\_rreq.c









aodv\_rreq.c

164-416：接收到的rreq消息的处理函数，rreq\_process()函数。

169~183：变量初始化。

189~190：忽略源自此节点的RREQ。否则缓冲我们自己发送的RREQ，就像我们对接收的那样。

196-201：检查data field 大小是否足够，不够则返回。

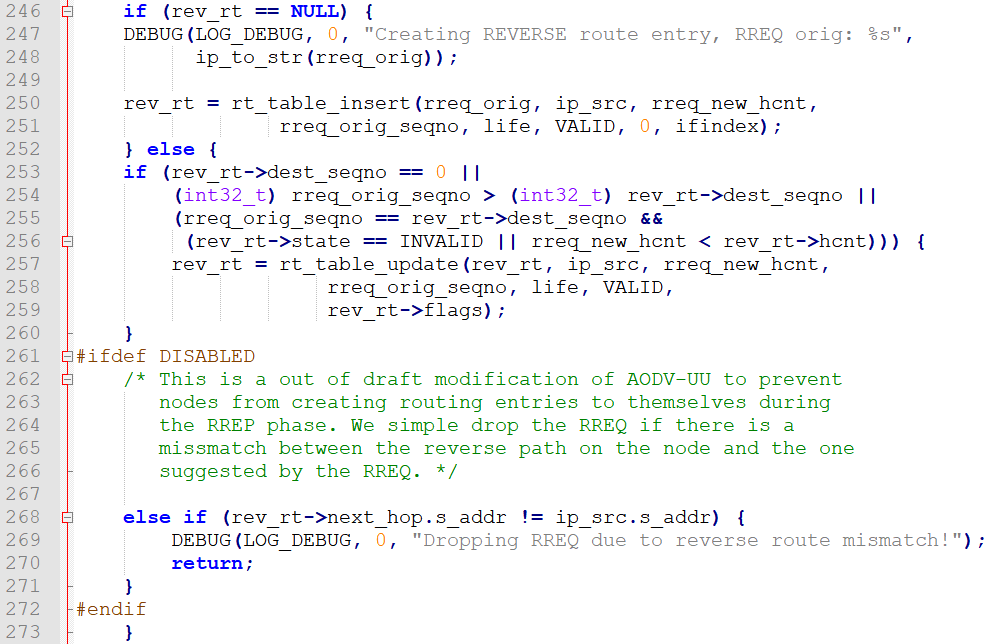
205-208：检查RREQ的前一跳是否在黑名单集中。如果是，则忽略RREQ。

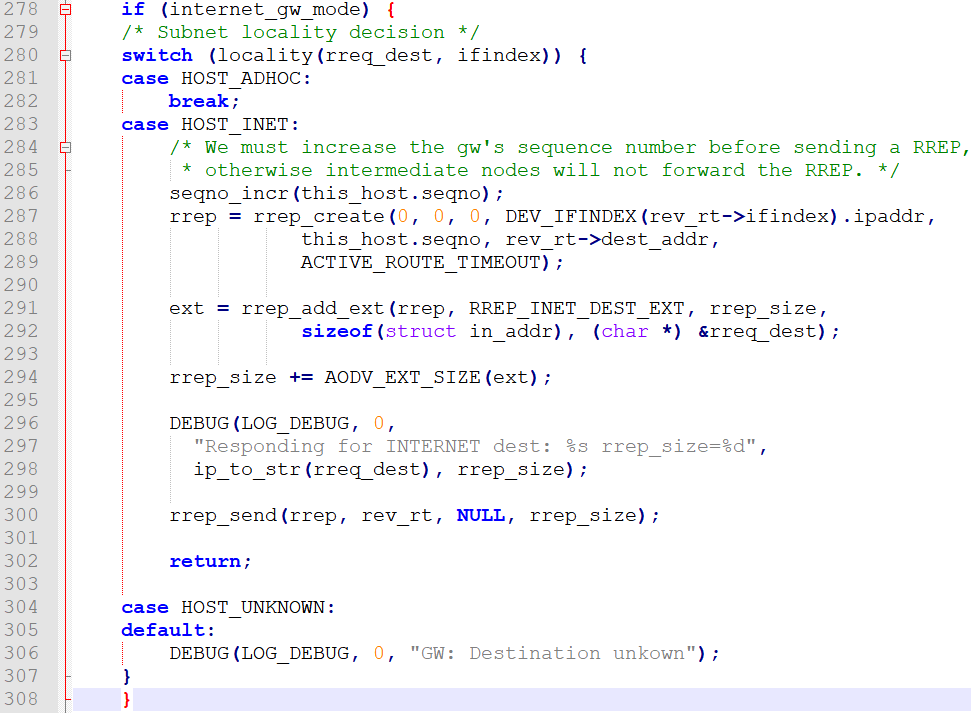
211-212：忽略已经处理的RREQ。

216:现在缓冲这个RREQ，这样我们就不会在PATH\_DISCOVERY\_TIME中处理类似的RREQ。

219-234：确定是否存在任何RREQ扩展。

aodv\_rreq.c





aodv\_rreq.c

241：从路由表中根据ip地址查找反向路由条目。

244：计算该路由条目生命期。

246-260：若找不到路由条目，则创建一条新的路由条目。若找到该路由条目，则根据序列号判断路由新旧，决定是否更新该路由条目。根据序列号判断路由信息新旧程度详见4.4。

至此，完成了中间节点维护指向路由源节点的反向路由。

接下来进行RREQ的转发或RREP的回应过程。

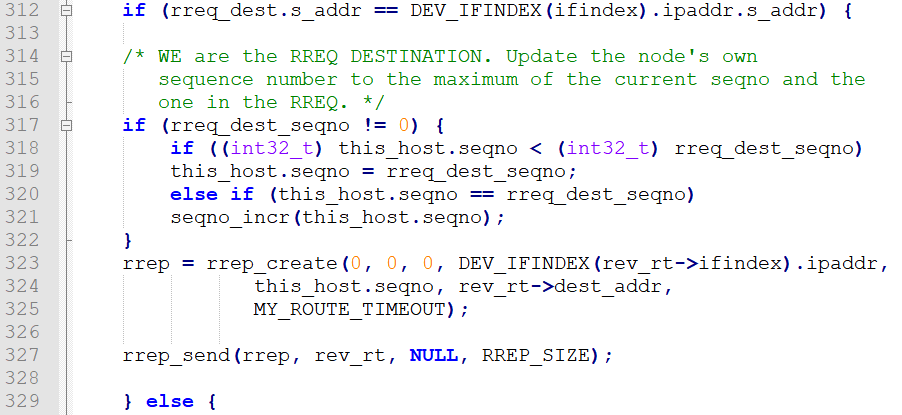
268-270：这是对AODV-UU的草稿修改，以防止节点在RREP阶段创建自己的路由条目。如果节点上的反向路径与RREQ建议的反向路径不匹配，则简单地删除RREQ。

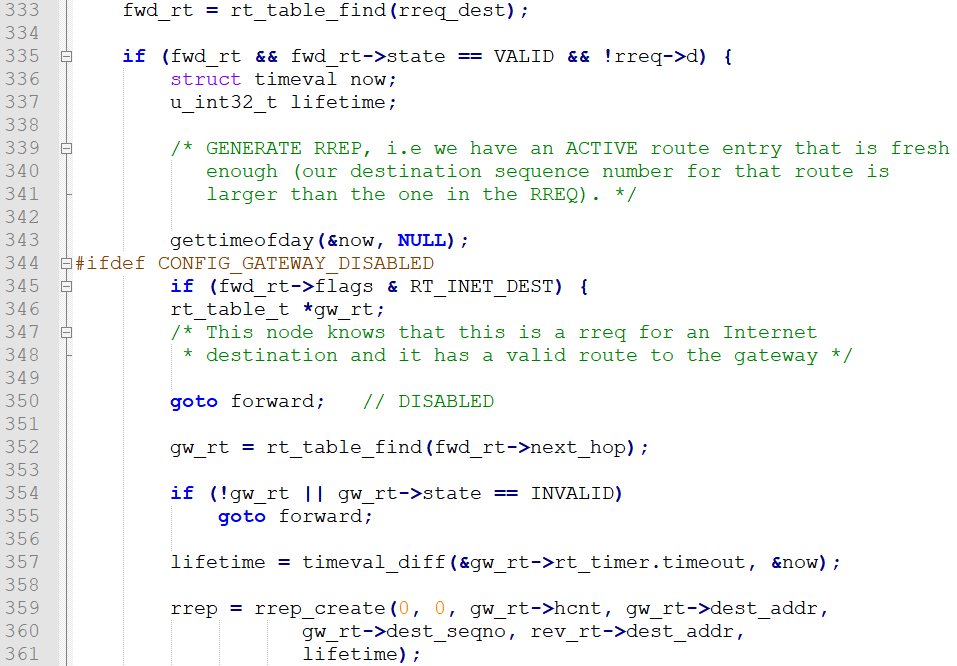
278-308：若开启了试验性的因特网网关支持，则进入如下代码。

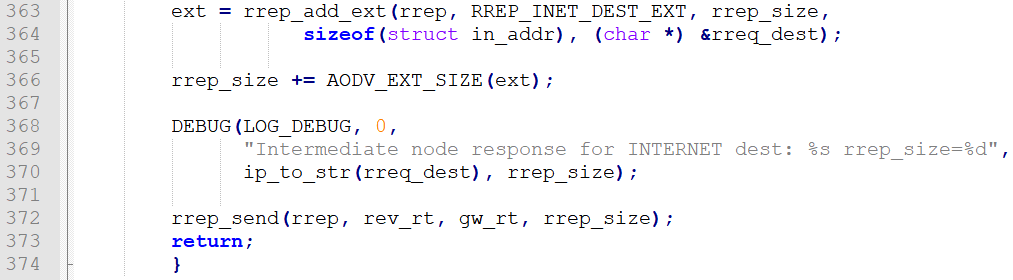
280：子网位置决策。

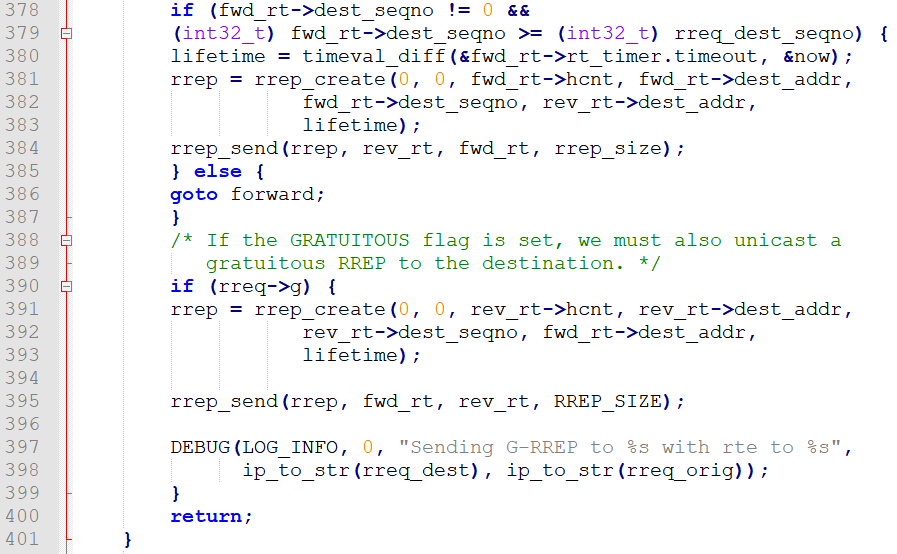
283-302：在子网位置是ipv4的时候，我们必须在发送RREP之前增加gw的序列号，否则中间节点将不转发RREP。

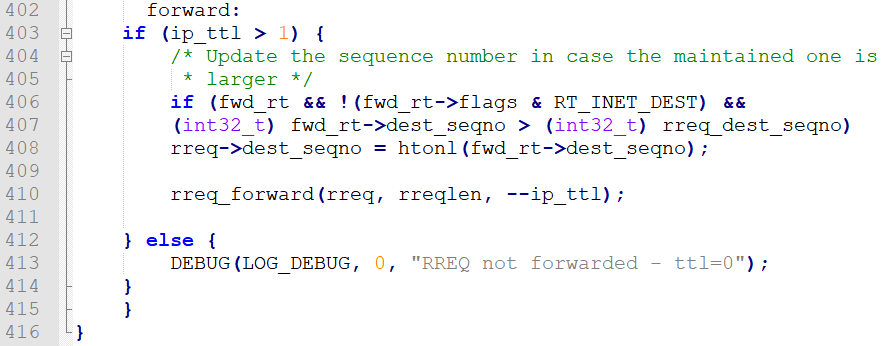
aodv\_rreq.c











aodv\_rreq.c

312-329：如果本地地址是RREQ的目的地，则应立刻发送RREP消息。由于本节重点在于中间节点转发RREQ消息，故暂且不分析。有需要可以查询4.2.4节。

329-415：如果我们不是目的地，则我们是一个中间节点。检查是否有活动路由条目，即是否有到RREQ目的地节点的路由表项，若有，则进入if代码区，进行进一步判断。若没有，则转发RREQ消息。

333：从路由表中查找是否有活动路由条目。

335~401：若有路由条目且该条目有效，则进入代码区，判断中间节点是需要转发RREQ消息还是发送RREP消息。

345~374：根据相关条件判断是否转发RREQ。

354~355：如果没有找到转发表项的下一跳或者下一跳无效，则进入RREQ转发。

359~372：否则创建并发送RREP消息。

378~386：判断接收到的rreq消息的序列号是不是足够的新，若足够新，则需转发RREQ

若不新，即过时消息，则创建并发送RREP消息。

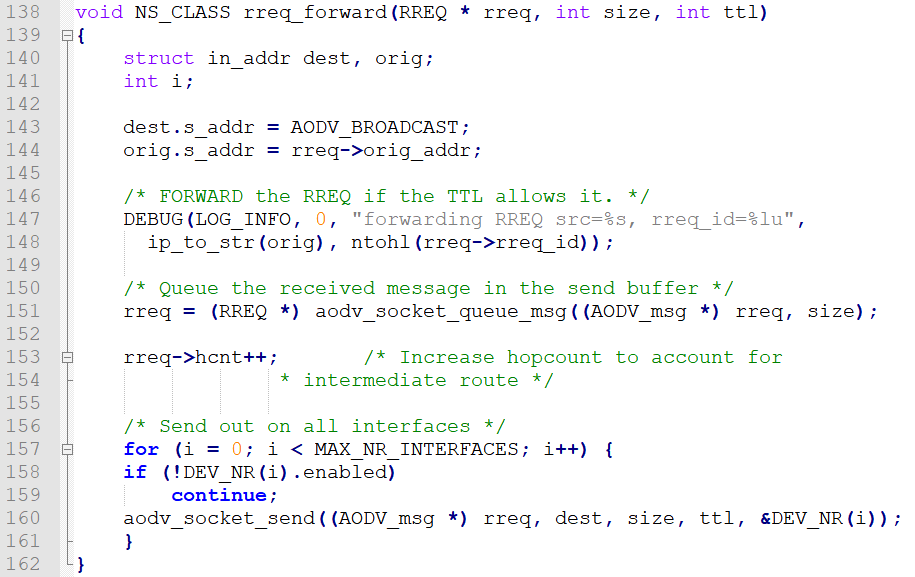
390~399：如果设置了免费标志，我们还必须将免费的RREP单播到目的地。

402~414：前面许多goto forward 语句的跳转地点，是在我们没有足够新鲜的活动条目，也就是不包含rreq目的节点的路由信息的时候，继续广播(转发)rreq消息。

410：调用rreq\_forward函数转发rreq消息。

rreq\_forward函数源码如下：

aodv\_rreq.c



aodv\_rreq.c

138-162：rreq消息转发函数rreq\_forward函数。

140-145：获取源地址和设置目的地址。

147-148：如果TTL允许，则转发RREQ。（ttl>0）

151：使再发送缓冲区接收的消息进入队列。

153：增加跳数以考虑中间路线。

157-160：在所有端口广播消息。

160：调用aodv\_socket\_send函数完成发送工作。由于前文已经对该函数进行过分析，故不再赘述。详见4.2.1。

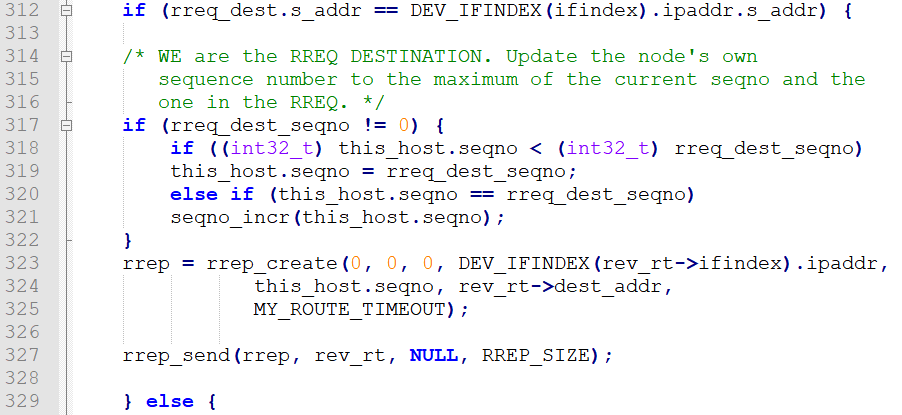
至此，完成了RREQ转发的工作。

### 4.2.4 产生RREP

目的节点或存在到目的节点有效路由的中间节点产生RREP路由应答帧。此节与上一节属于并列关系。是节点在收到RREQ消息后走的不同分支。故相同代码不在罗列，避免冗余。

若收到RREQ的节点为目的节点，则执行如下代码：

aodv\_rreq.c



aodv\_rreq.c

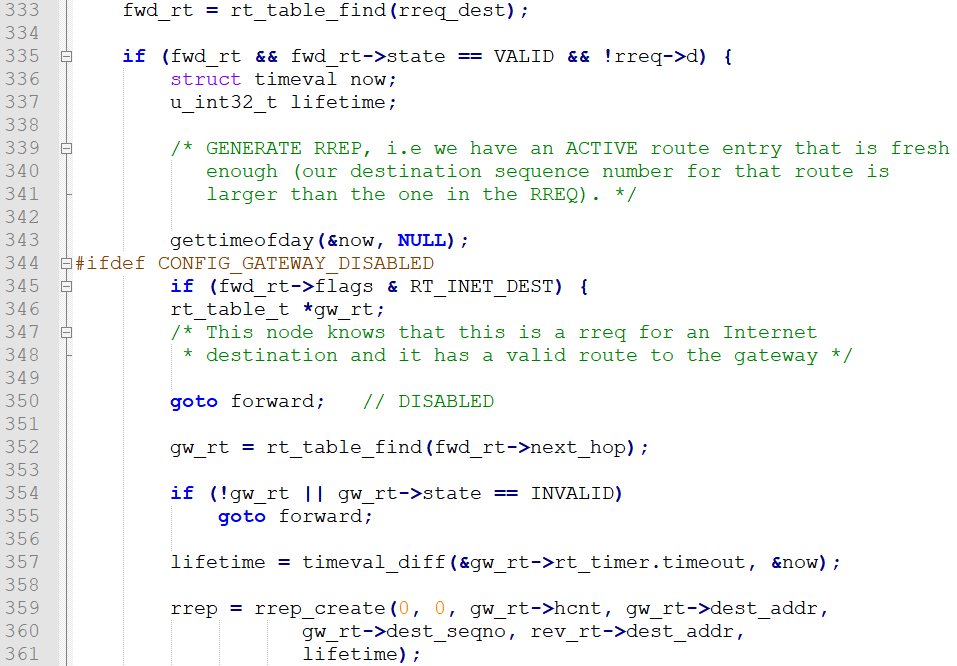
312~329：如果本节点是目的节点，则沿发送路径发送RREP消息。

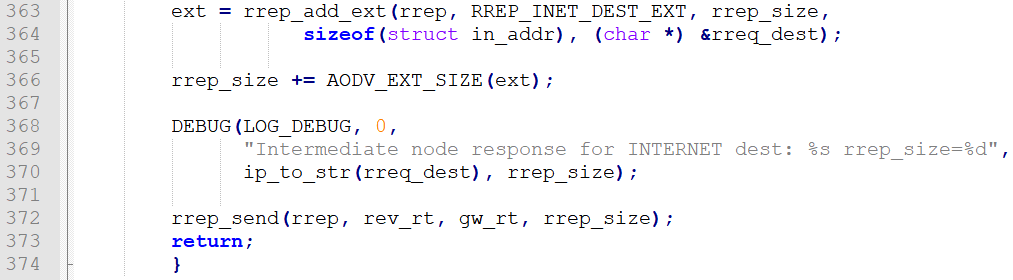
317~322：我们是RREQ的目的地。将节点自身的序列号更新为当前序列号和RREQ中的序列号的最大值。用于帮助其他节点判断路由消息新旧。详见4.4节。

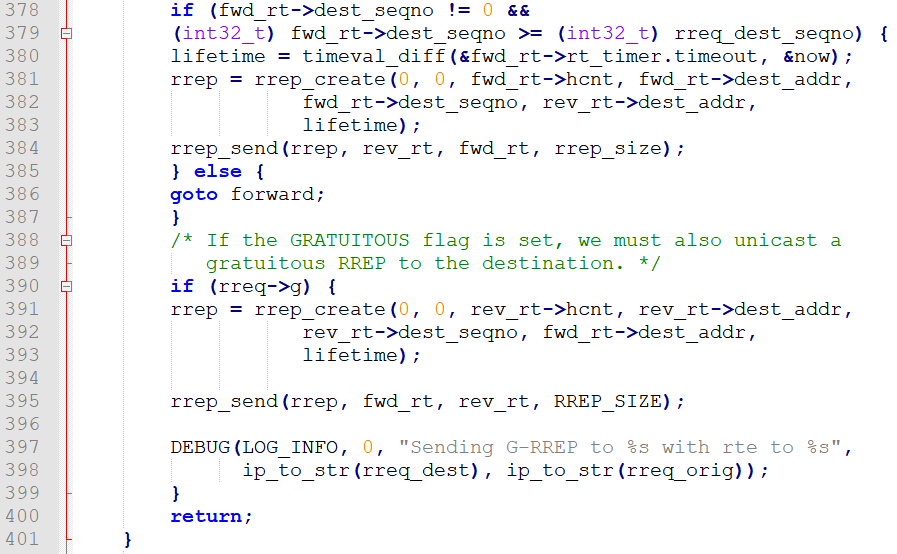
323~327：创建RREP消息，并调用rrep\_send函数，发送RREP消息。

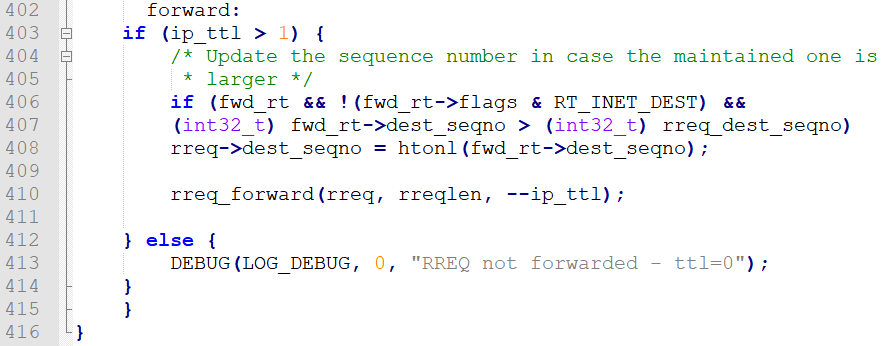
除了目的节点会发送RREP消息，存在到目的节点有效路由的中间节点也会产生RREP路由应答帧。其相关代码如下：

aodv\_rreq.c









aodv\_rreq.c

333：从路由表中查找是否有活动路由条目。

335~401：若有路由条目且该条目有效，则进入代码区，判断中间节点是需要转发RREQ消息还是发送RREP消息。

345~374：根据相关条件判断是否转发RREQ。

354~355：如果没有找到转发表项的下一跳或者下一跳无效，则进入RREQ转发。

359~372：否则创建并发送RREP消息。

378~386：判断接收到的rreq消息的序列号是不是足够的新，若足够新，则需转发RREQ

若不新，即过时消息，则创建并发送RREP消息。

390~399：如果设置了免费标志，我们还必须将免费的RREP单播到目的地。

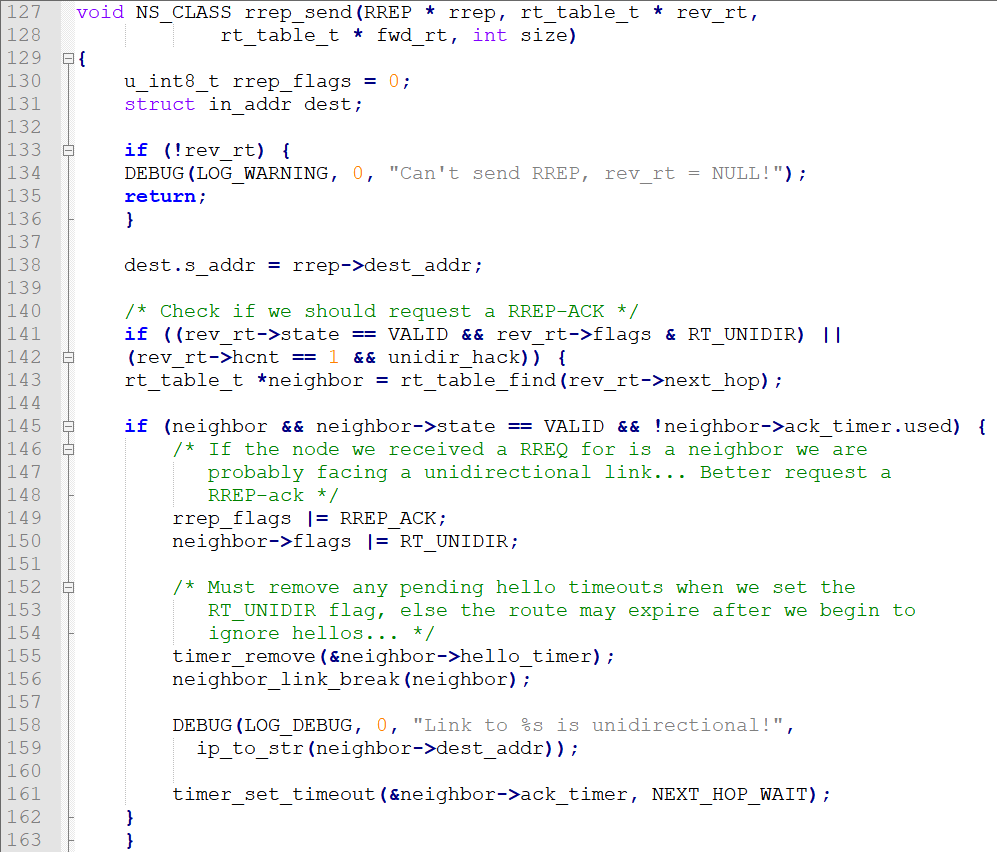
RREP消息发送过程需调用rrep\_send函数，详见下节；

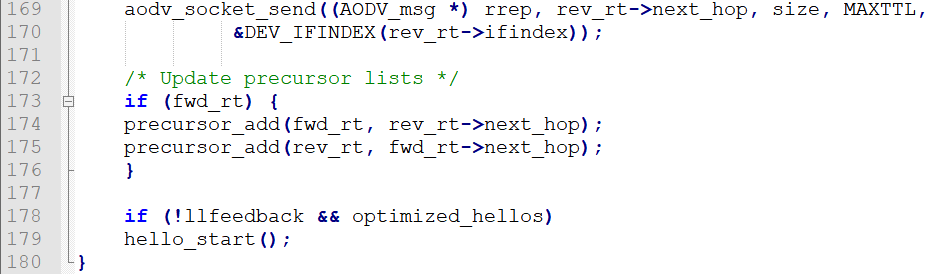
### 4.2.5 RREP单播至源节点

RREP通过之前建立的反向节点单播至源节点。

rrep\_send函数用于发送rrep消息。其源码如下：

aodv\_rrep.c





aodv\_rrep.c

127~180：rrep消息发送函数rrep\_send。

133~135：如果当前路由表项为空则不能发送rrep消息。

141~163：检查是否应该发送rrep\_ack消息。

143：从路由表中查找rreq消息目的地址的下一跳。

145~162：如果接收到的rreq消息目的地址的IP地址是当前节点的邻居节点，则可能面临这样单项链路这样一种尴尬的情形，所以应该首先发送一个rrep\_ack消息。

155~161：因为当我们决定忽略hello消息时，路由将会终止，所以当设置单向路由时，必须移除期间内的所有hello计时器。

165~167：更新日志记录信息。

169：调用aodv\_socket\_send函数发送RREP消息。

173~175：更新fwd\_rt和rev\_rt的先驱路由表。

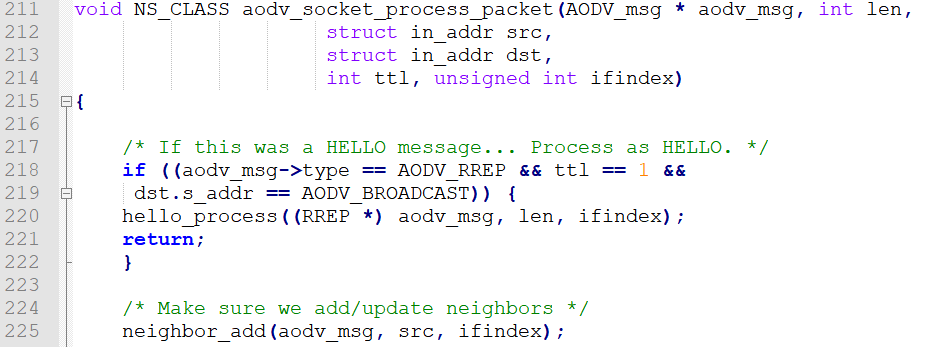
由于前文已经介绍过aodv\_socket\_send函数，这里不再赘述。

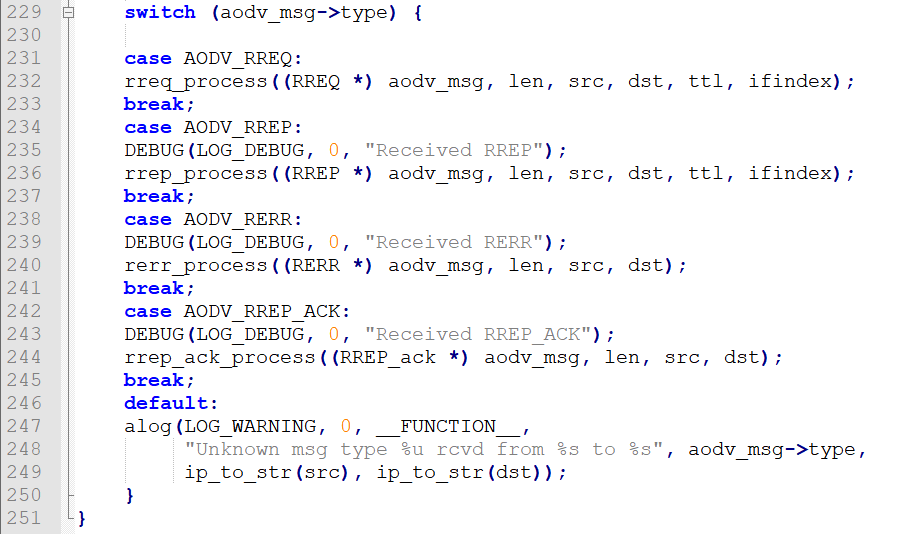
### 4.2.6 收到RREP

若源节点收到RREP应答帧，至此完成了路由发现过程，源节点可以向目的节点发送数据包。若中间节点收到RREP应答帧，则会单播RREP至路由的下一跳。

代码如下：

aodv\_socket.c





aodv\_socket.c

211-251：aodv\_socket\_process\_packet函数的声明，此函数负责辨别接收到的包的类型，并且对不同类型的包进行不同处理，此段代码负责此函数的实现。

218-222：如果包是一个HELLO消息，就把控制传递给hello\_process函数。

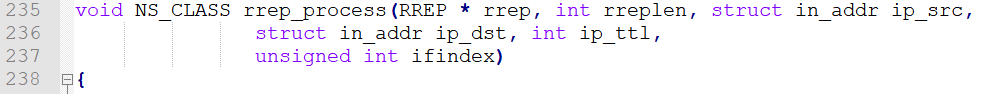
225：根据消息所包含的信息添加或更新邻居节点的信息。

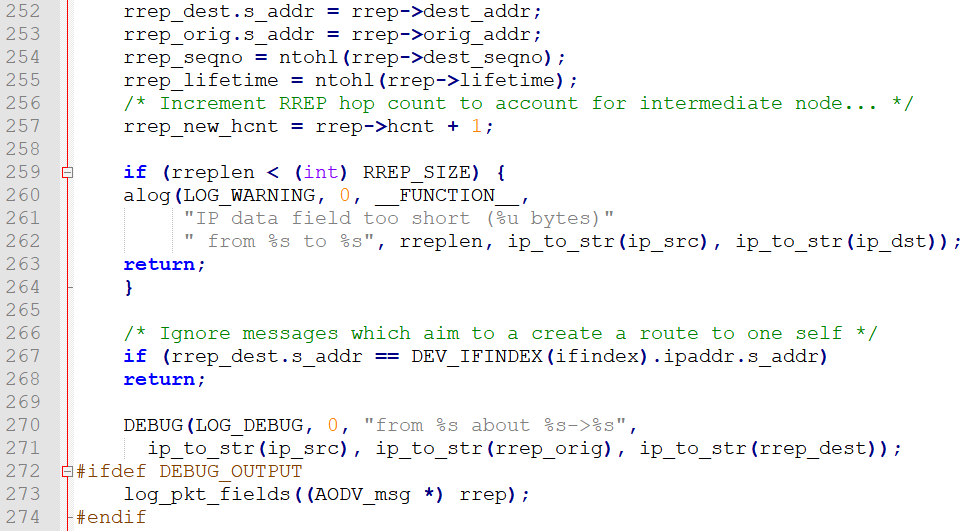
229-250：如果是HELLO以外的消息，根据类型记录路由日志并处理收到的消息。

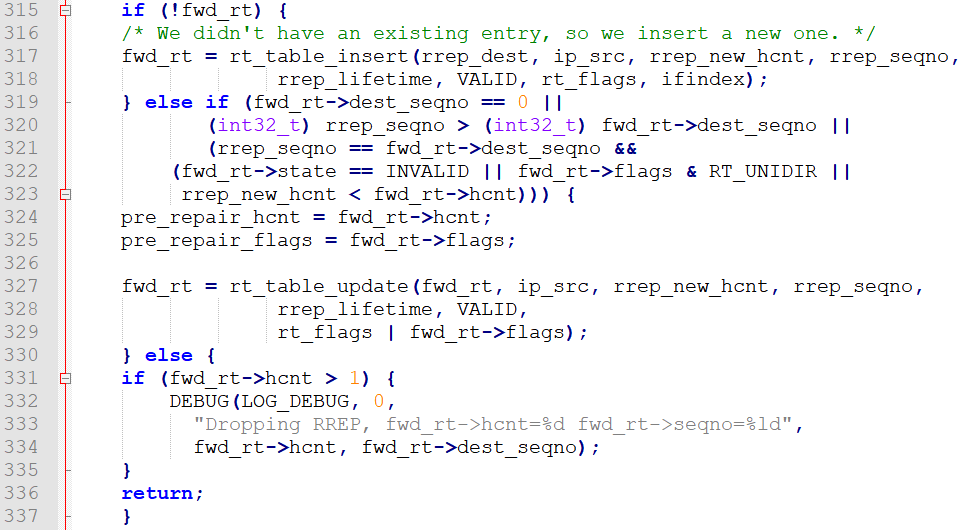
在234~237行中，该函数调用了rrep\_process函数，处理rrep消息。

rrep\_process函数代码如下，由于篇幅过长，这里仅截取核心部分：

aodv\_rrep.c







aodv\_rrep.c

235-411：rrep消息处理函数rrep\_process()。

252-255：在受影响的字段上转换为正确的字节顺序，便于处理。

257：增加RREP跳数以考虑中间节点。

259-264：判断收到的rrep消息是否小于下限值，若成立则立即返回。

267-268：忽略那些旨在创建通往自我的路径的消息。

277-308：确认是否存在扩展数据，并且根据情况处理。

312-337：检查我们是否应当转发路由。

315-318：如果我们没有当前rrep的目的地在路由表中，那么我们就在路由表上加上这一项。

319-330：如果我们有当前rrep目的地在路由表中，则根据序列号判断新旧，若表中以过时，则更新表项。

330~337：如果是新的，则日志记录。

342-350：如果设置了RREP应答标志，我们必须向应答的目的地发送一个RREP确认。

346：调用aodv\_socket\_send函数，发送RREP确认。

354-400：检查此RREP是否适用于我们（即，我们是否这这个RREP的目的地，即是否是源节点）

363~373：添加一个“假”路由，表明这是一个Internet目标，因此应该被封装并通过网关路由。

382~399：如果路由以前处于修复状态，则应向路由的源发送一个不删除的RERR，以便它可以选择重新启动目标的路由发现。修复了一个错误，导致修复标志被取消设置，并且RERR从未被发送。

400-407：我们不是源节点，我们是中间节点。若rrep的目的地有效，则转发该rrep消息，否则记录日志。

至此，完成了AODV协议的路由发现过程。在源节点和目的节点，及中间节点都建立起了一条路径。

## AODV路由维护

1. Hello消息：Hello消息帧就是TTL=1时的REEP帧。
2. Hello消息帧用于检测活跃路径上相邻节点的链接状况
3. 只有当某节点位于某活跃路径之上时，他才能发送Hello消息帧
4. 活跃路径节点以HELLO\_INTERVAL为周期发送Hello消息
5. 在DELETE\_PERIOD的时间内没有收到来自邻居节点的Hello消息，则认为该链路失效
6. 发起一次指向该邻居节点的局部修复
7. 路由修复超时以后，路有错误信息RERR向源节点和目的节点发送
8. RERR在传播过程中，各中间节点删除该失效路径上相应的路由信息

### 4.3.1 Hello消息

Hello消息：Hello消息帧就是TTL=1时的REEP帧

Hello消息帧用于检测活跃路径上相邻节点的链接状况。

只有当某节点位于某活跃路径之上时，他才能发送Hello消息帧。

### 4.3.2 发送Hello消息

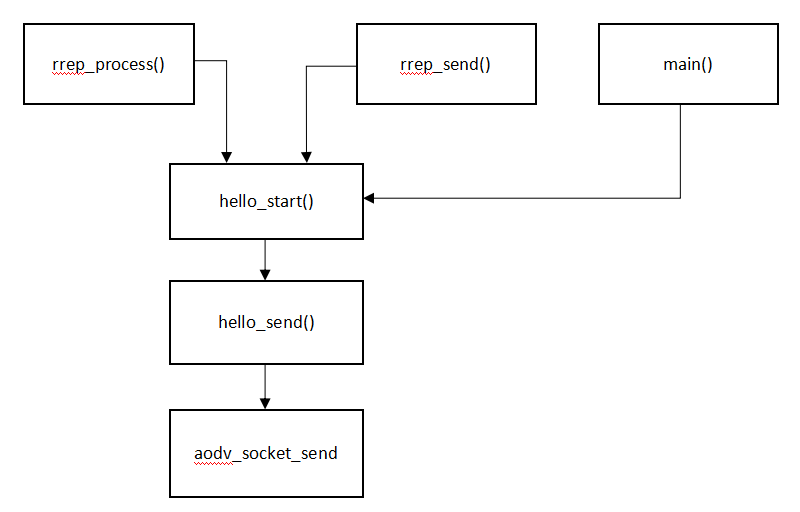
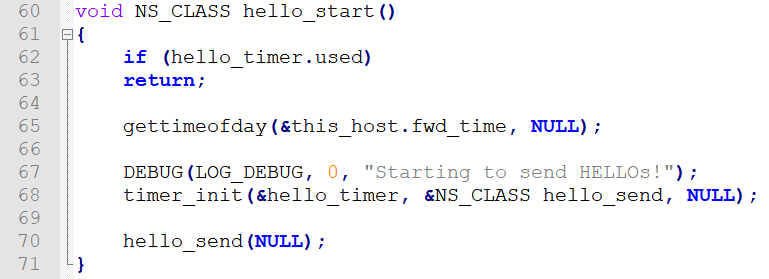


图4.4 hello消息的发送的函数调用关系

活跃路径节点会未开启链路层反馈，开启-o（只在转发数据包时发送hello）的情况下，以HELLO\_INTERVAL为周期发送Hello消息。其函数调用关系如图4.4所示：

下面依次介绍相关代码。

aodv\_hello.c



aodv\_hello.c

60-71：发送hello消息的准备函数hello\_start。

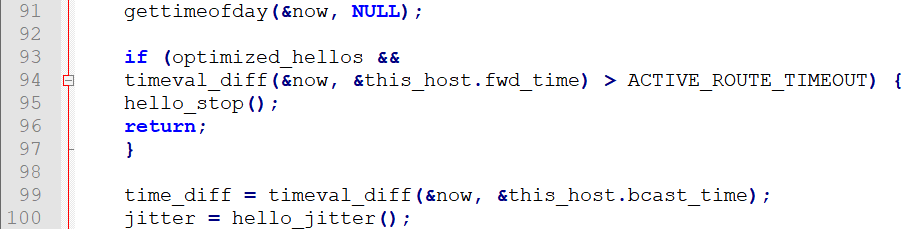
62-63：检测hello\_timer计时器是否开启，若开启表明hello消息已发送，则直接返回。

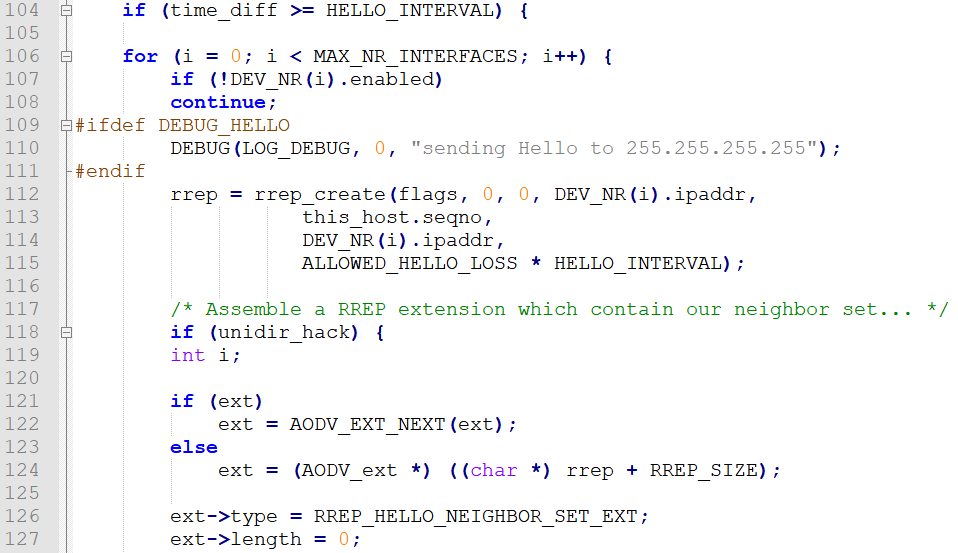
67-70：在Debug.h中宏定义的DEBUG函数第三个参数设置输出信息"Starting to send HELLOs!"表示此时准备要发送hello消息。随后调用的timer\_init函数的传入三个参数，分别是&hello\_timer：启动hello\_timer计时器；&NS\_CLASS hello\_send：hello\_send；传入数据NULL。随后即执行hello\_send函数。

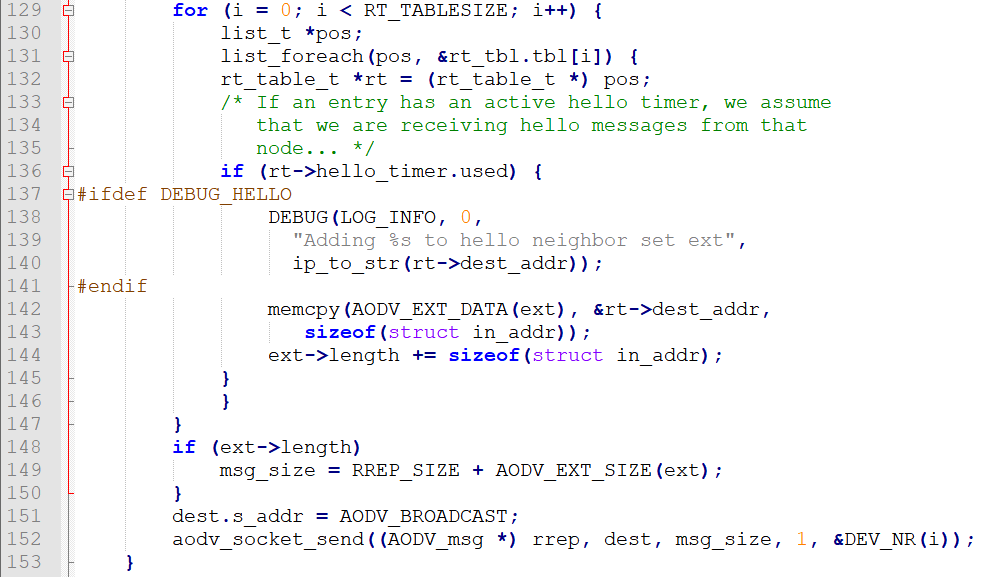
hello\_send函数源码如下，由于篇幅过长，这里仅展示核心代码：

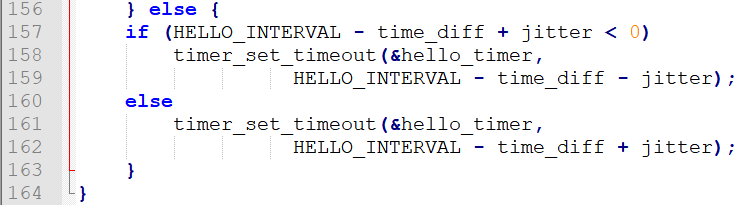
aodv\_hello.c











aodv\_hello.c

80-164：hello消息发送函数hello\_send。

91：gettimeofday()：得到当前时间和时区。

93-96： 判断若当前时间和最后一个hello消息发送的时间差大于路由超时时间，则可以确定当前节点没有任何一条活跃路径与相邻节点相邻，因此执行hello\_stop()。

99-100：获取当前时间和最后一个广播分组发送时的时间差。

104：判断第99行计算的时间差与发送hello消息的周期HELLO\_INTERVAL的大小，保证如果在HELLO\_INTERVAL周期内已经发送hello消息的情况下不会重复发送消息。

109-115： 程序判断如果宏定义了 DEBUG\_HELLO 则显示广播 hello 消息。

112：创建了 hello 消息。

129~147：遍历路由表，如果一个条目有一个活跃的hello计时器，我们假设我们正在从该节点接收hello消息。

151-152：调用aodv\_socket\_send函数广播这个ttl为1的rrep消息(hello消息)。

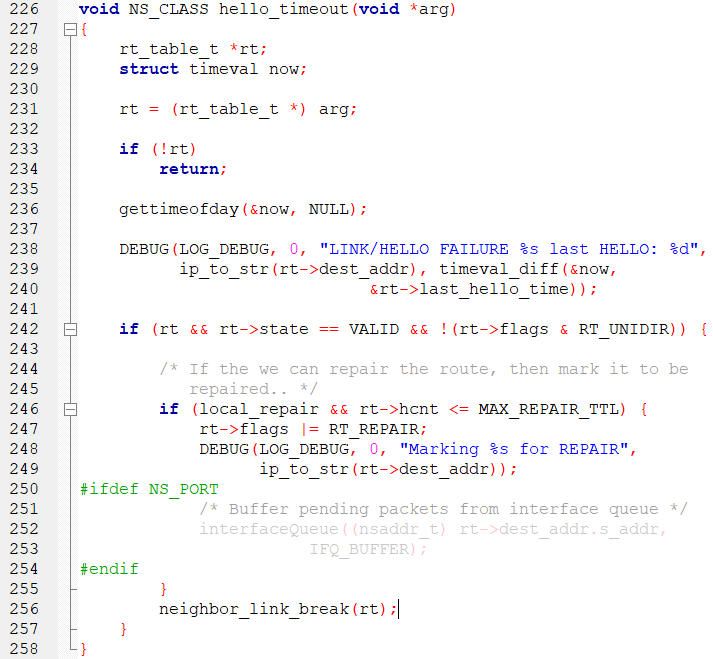
155：计时器超时设置。

156-163：若时间差比hello消息发送周期小的时候，不进行发送。

### 4.3.3 链路失效判断

在DELETE\_PERIOD的时间内没有收到来自邻居节点的Hello消息，则认为该链路失效；

aodv\_timeout.c



aodv\_timeout.c

233-234：边界检查，输入路由是否存在，不存在则返回。

236：在变量now中存下当前时间。

238-240：日志记录下最后一条来自于当前路由的HELLO消息的时间。

242-249：如果可以修复此条路由，那么设置其标志位为REPAIR。

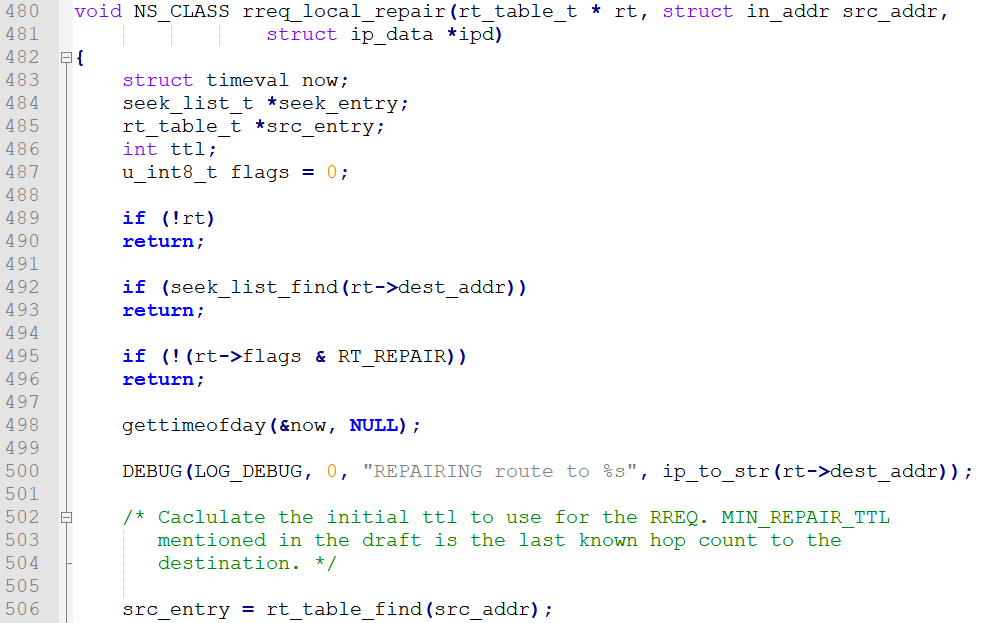
250-254：如果定义了宏NS\_PORT，则执行此段代码，意思是要缓冲从接口队列挂起的数据包。

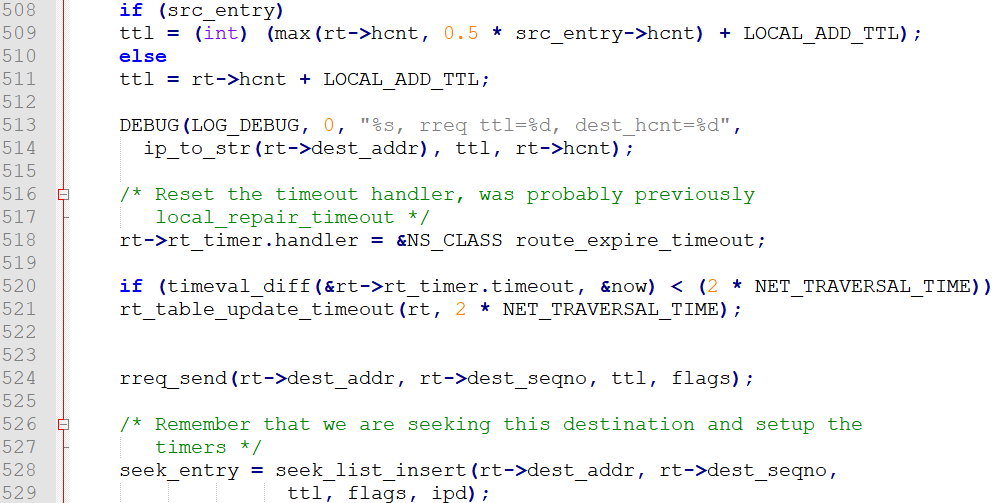
256：然后设置这条路由为断开。

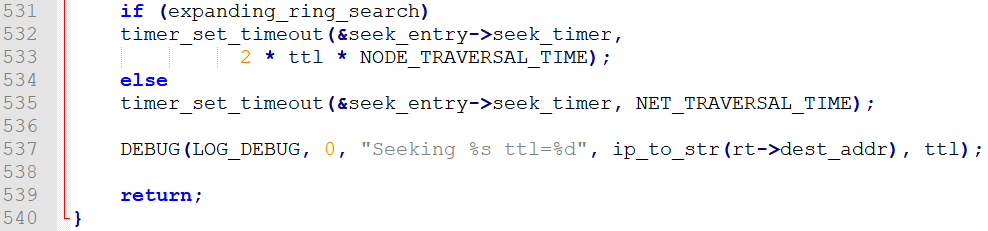
### 4.3.4 局部修复

判断链路失效后，会发起一次指向该邻居节点的局部修复。代码如下：

aodv\_rreq.c







aodv\_rreq.c

482-540：本地修复函数，rreq\_local\_repair函数，它与路由发现函数非常相似。

483-498：初始化变量。

506-511：计算用于RREQ的初始TTL。

518-524：重置超时处理程序。

524：发送RREQ消息，用于重建路由路径。

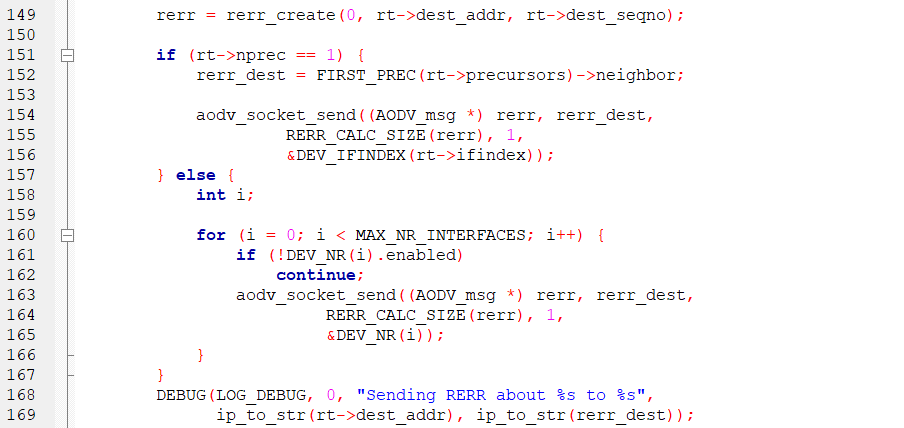
528-537：记住我们正在寻找这个目的地并设置计时器。

### 4.3.5 修复超时，发送RRER路由错误信息

路由修复超时以后，路有错误信息RERR向源节点和目的节点发送，

aodv\_timeout.c





aodv\_timeout.c

149：创建一个rerr消息，并广播通知其他节点。

151-156：若rt->nprec == 1， 则让rerr消息发往邻居。

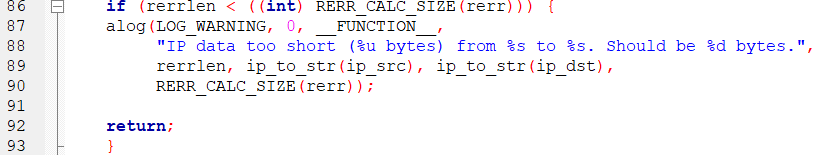
157-166：若rt->nprec ！= 1，广播发送。

168-170：日志记录rrer消息发送情况。

### 4.3.6 中间节点处理RRER

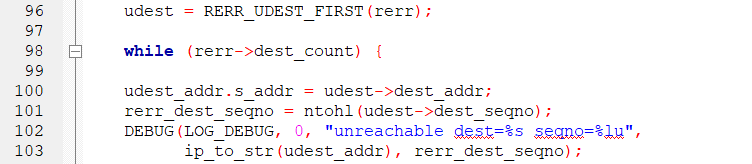
RERR在传播过程中，各中间节点删除该失效路径上相应的路由信息。





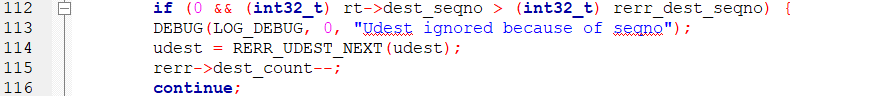
aodv\_rerr.c

86-92：判断接收参数rerrlen，即rerr消息的大小，若其小于下限值RERR\_CALC\_SIZE(rerr),则输出警示消息并返回。



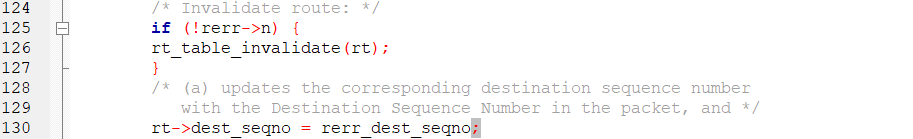
aodv\_rerr.c

96-103：检查不可达目的节点，并循环输出其IP地址以及序列号。



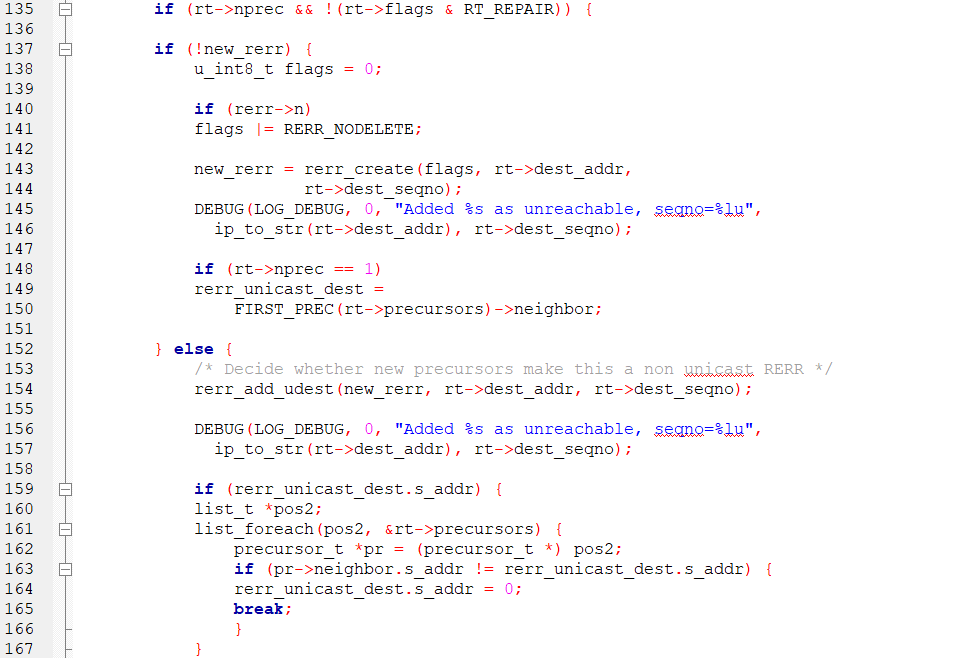
aodv\_rerr.c

112-116检查rerr消息中不可达节点的目的序列号与路由表项中保存的最新相应节点的序列号，若后者大于前者说明消息已过期，直接返回。



aodv\_rerr.c

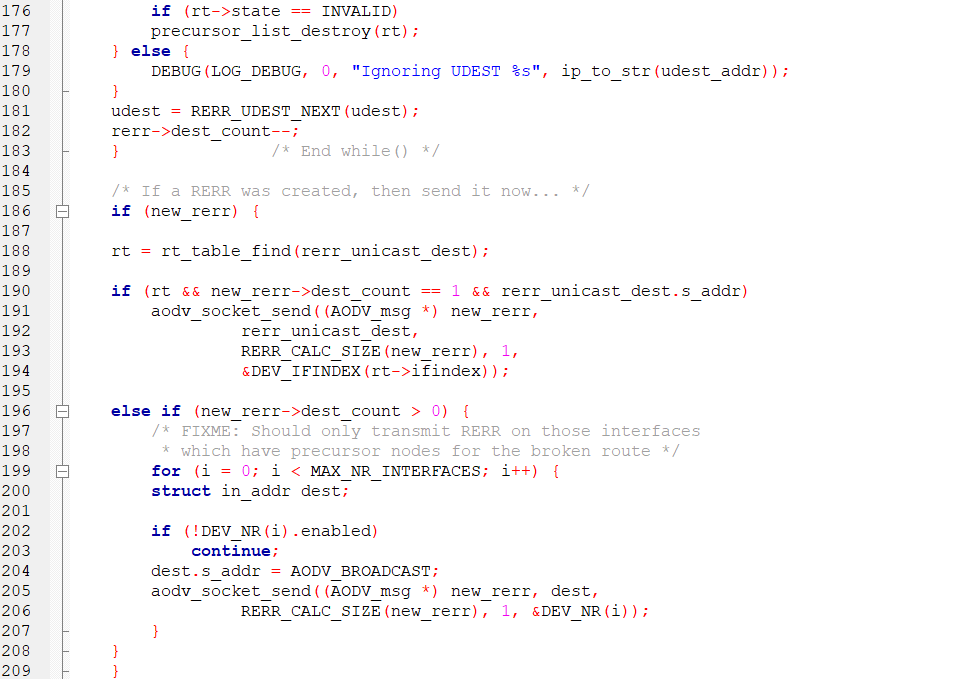
124-130：调用rt\_table\_invalidate()使路由rt无效，随后将rt的目的节点序列号更新为消息包中相应的目的节点序列号。

 aodv\_rerr.c

135-150：半段先驱列表是否为空，若为空，则在rerr消息添加目的IP为不可达节点。

154-167：决定新的前驱列表是否使这个成为非单播RERR，若pr的邻居节点地址不是rerr单播地址，清空rerr单播地址。

aodv\_rerr.c

 aodv\_rerr.c

176-183：为所有不可达目的节点删除先驱路由列表。

186-209：如果构造的rerr消息的dest\_count=1，那么现在即单播rerr消息；否则，我们只应该向包含断裂路由先驱节点的端口发送rerr消息。

## AODV路由信息新旧判断

1. AODV依赖网络中每个节点维护自身的序列号。
2. 源节点在广播路由请求帧RREQ之前要更新自己的序列号，即将序列号加1。
3. 目的节点在产生RREP应答帧之前也要将自身的序列号加1。
4. 每个节点在对各自的序列号加1的时候，是将其视为无符号数进行的。
5. 通过比较来自目的节点路由控制帧中的序列号SN1和本节点维护的目的节点的序列号SN2，就可以确定本链路的新旧程度。如果SN2-SN1<0(有符号数相减)，说明路由表中的维护信息已过时，应将路由信息更新至路由控制帧最新的路由信息。

### 4.4.1 源节点更新序列号

源节点在广播路由请求帧RREQ之前要更新自己的序列号，即将序列号加1，每个节点在对各自的序列号加1的时候，是将其视为无符号数进行的。

aodv\_rreq.c





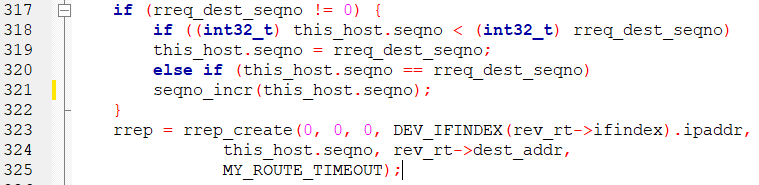
aodv\_rreq.c

78-79：在节点发起RREQ泛滥之前，它必须立即增加它的序列号。

### 4.4.2 目的节点更新序列号

目的节点在产生RREP应答帧之前也要将自身的序列号加1，每个节点在对各自的序列号加1的时候，是将其视为无符号数进行的。

aodv\_rreq.c

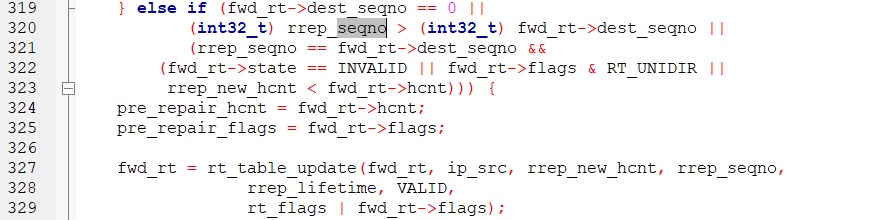


aodv\_rreq.c

317-322：如果我们（也就是目的地）的序列号不为0，当我们和目的序列号不同的时候，取大的序列号，当我们和目的的序列号相同的时候，序列号+1。

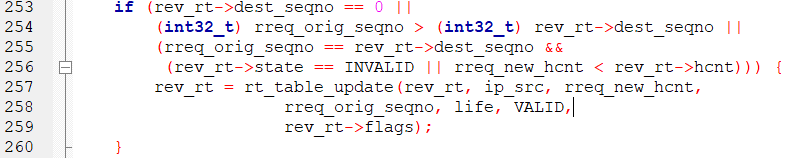
### 4.4.3 对比序列号判断路由消息的新旧

通过比较来自目的节点路由控制帧中的序列号SN1和本节点维护的目的节点的序列号SN2，就可以确定本链路的新旧程度。如果SN2-SN1<0(有符号数相减)，说明路由表中的维护信息已过时，应将路由信息更新至路由控制帧最新的路由信息。



aodv\_rrep.c

319-329：在rrep消息发送的目的序列号大于本节点保存的目的序列号的时候，更新本节点的路由表，使用收到的rrep包更新。



aodv\_rreq.c

253-260：在rreq消息发送的源序列号大于本节点保存的源序列号的时候，更新本节点的路由表，使用收到的rreq包更新。

## 拥塞控制

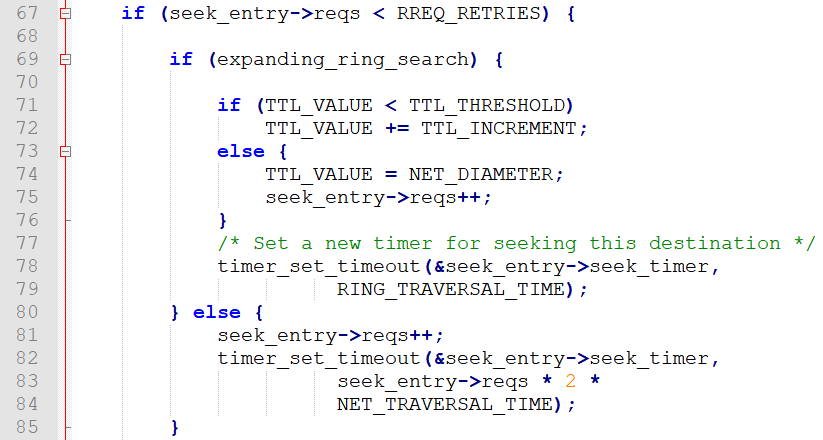
可用于AODV协议的一种拥塞控制方式如下

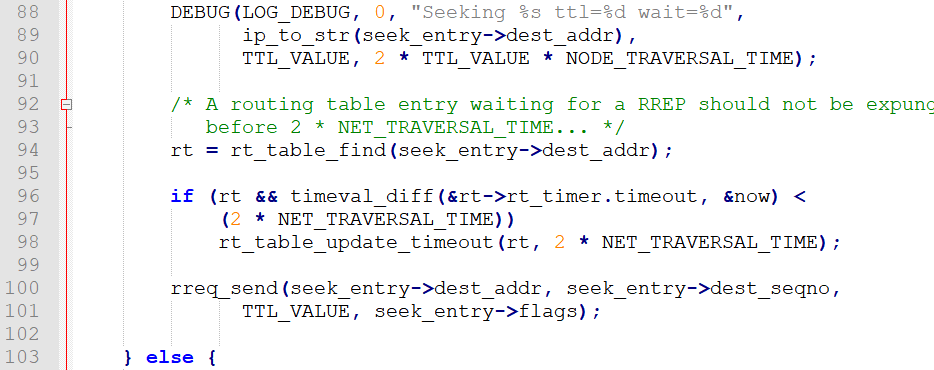
* 源节点在发送RREQ后，在规定的时间内没有收到来自目的节点的RREP时，他可以选择再次发送RREQ路由请求帧。在尝试了RREQ\_RETRIES次之后，如果依旧收不到RREP，则在路由表中标记该目的节点不可达，并通知应用层。
* 每次在重新发送RREQ请求帧时，等待RREP应答帧的时间要在原来时间的基础上乘以2，避免拥塞。

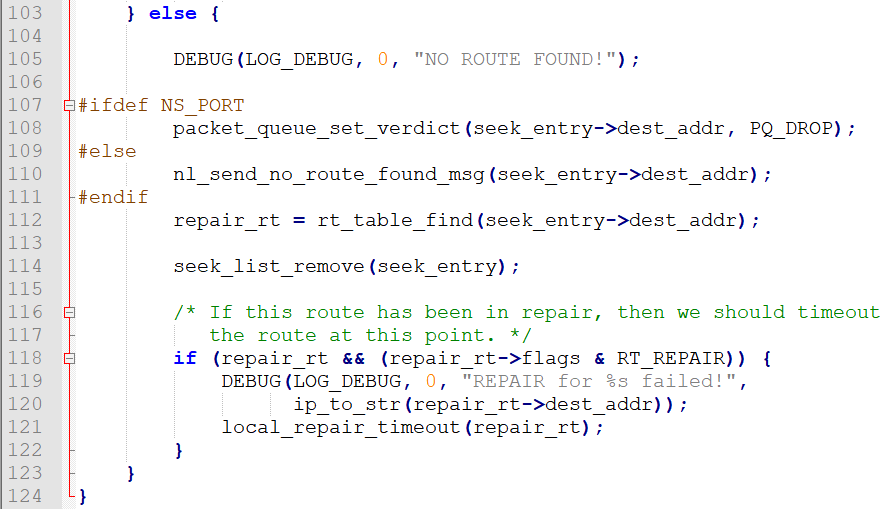
代码如下所示。

aodv\_rreq.c









aodv\_rreq.c

67~103：源节点发送RREQ后，规定时间内没有RREQ消息返回，如果此时尝试次数小于RREQ\_RETRIES，则可以再次发送RREQ帧。

69~80：若开启拓展环搜索法，则使用该方法寻找。

94：在路由表中查找目的表项。

96~101：每次在重新发送RREQ请求帧时，等待RREP应答帧的时间要在原来时间的基础上乘以2，避免拥塞。

101：发送新的RREQ数据帧。

103~123：由于仍旧收不到RREP，则在路由表中标记该目的节点不可达，并通知应用层。

# 第五章 总结

AODV 协议作为一个无线组织网络中的路由协议，其设计精巧，结构简明，

容易实现。在分析过程中，我们感受到，这一协议运用直接简洁的方法，解决了

复杂的问题。这一化复杂为简单的思想很有价值。

分析代码的过程中，可以看出，代码的架构非常清晰，风格严谨，注释

清楚。程序中有大量的 if 语句结构进行错误判断，这样使代码鲁棒性很强。此

外，各个函数的功能也非常清楚，命名规范，各模块之间很好的体现了“高内聚，

低耦合”的程序设计原则。通过阅读以及分析AODV的代码，我们真的学到了很多，为我们的以后的代码整体布局与编写都做到了一个极好的模范作用。

随着技术的不断革新，无线网络的应用越来越广泛。这正成为研究的热门

领域。AODV 协议的出现，解决了无线自组织网络的按需路由问题。在无线个域

网中，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 的报文设计并没有考虑这些问

题。希望在今后能有可能进行改进。