网络协议栈分析与设计课程大作业

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
201792002	仲点	软网 1702	1．协议过程 2．内核路由及队列维护 3．Netfilter 处理 4．内核与用户交互 5．协议系统框架分析 6．排版	
201792437	崔国繁	软网 1702	1．Sock 模块 2．路由信息处理 3．路由代码分析 4．总结	
201792272	付汉民	软网 1702	1．用户层路由表处理 2．超时管理模块 3．文献搜集	

大连理工大学

Dalian University of Technology

目　　录

1 引言

1.1 概 述

路由协议是 Ad-Hoc 网络研究的热点和难点问题，近些年来，人们已经提出 20 种以上的 Ad Hoc 网络路由协议，目前大多数的 Ad Hoc 网络路由协议都是在理论分析和模拟仿真环境下进行研究，真正实现的并不多。AODV 按需距离向量路由协议 (Ad hoc On-demand Distance Vector Routing)，简称 AODV 路由协议，是一种典型的按需的路由协议。我们大作业的内容是基于 RFC3561 完成对 AODV 协议的学习，并使用上课学到的代码方法分析 AODV 的实现之一 AODV-UU-0.9.6。AODV 协议主要有一下几个特点:

1、 允许节点迅速获得目的地的路由，并不需要维护超出自己辐射范围的路由。
2、 允许节点及时地相应网络拓扑的损坏和变化，它的运行时自由循环的。
3、 通过避免 Bellman-Ford 无穷计算的问题，提供快速的收敛。
4、 当发生链路损坏是，能够通知受影响的节点，以便其使用丢失连接机制来使相关路由失效。

1.2 分工

姓名	班级	学号	完成工作
仲点	软网 1702 班	201792002	阅读 RFC3561; 文档排版; 内核与用户层交互; 全局变量; 协议过程讲解; 参考文献搜集
崔国繁	软网 1702 班		
付汉民	软网 1702 班		

1.3 AODV-UU 介绍

AODV-UU 是由瑞典乌普萨拉大学与爱立信公司基于 GPL 联合发布的 Ad Hoc 网络路由协协议，是一种最成熟的 AODV 实现程序。该协议是基于 RFC3561 实验版本的改进版本。

AODV － UU 实现 AODV 路由协议的主要目的是为了试验研究无线移动网络的按需路由协议，并且在 RFC3561 的 AODV 协议草案基础上增加了一些其他功能。例如 Hello 消息增加了一些功能，以及选择单向链接监测和冲突避免。在 802.11 标准中使用 HELLO 广播信息会引起链路检测的一些困难。因为在 802.11 标准中广播信息传输速度低于单播信息的速度，并且广播的范围大于单播，鉴于此点，在单播信息失败后节点可能还认为链路是动态活动的，这样会引起报文错误率增加，AODV － UU 实现方法对此

问题进行了补偿。另外还提供了 Internet 网关和多种网络接口支持等。

2 AODV 路由协议

AODV 路由协议包括路由请求、路由应答和路由维护 3 个过程，依赖 RREQ(路由请求)、RREP(路由应答)、RERR(路由错误) 及 HELLO 共四类报文。各节点维护路由表分别对不同报文进行处理，维护路由信息的正确有效。

2.1 基本过程

路由请求，当源节点向目标节点发送数据时，先在路由表中查找去网目标的路由表条目，如找到则选用该路由进行传输；否则够造 RREQ 报文并广播寻找去往目标的路由。其他节点接收到该 RREQ 后，先判断自身是否为目标节点。若是则阐述 RREP 报文并沿反向路径返回给请求节点；否则查找路由表以确定是否有去往目标节点的有效路由条目，有则向反向邻节点返回 RREP，无则继续广播 RREQ。

路由响应，节点收到 RREQ 后，发现自身为目标节点或自身路由表中有去往目标的有效路由，则产生 RREP 并沿反向路径返回给请求节点。当上游节点接收到 RREP 后，先建立到目标节点的路由，并查看自身是否为请求节点，若是则停止转发，否则继续转发 RREP。

路由维护。路由中断时，节点先启动本地路由修复，若无效，则向相关邻节点发送 RERR，告知路由中断。此外，路由器缓存及时器会周期性地把超时的路由条目从路由表中删除，邻居节点计时器也会周期性地广播 HELLO 报文检测节点的连通性，及时清除中断路由。

2.2 路由信息格式

2.2.1 路由请求报文

RREQ 路由请求报文包括以下几个部分

1、 类型：控制报文的类型，RREQ 为 1
2、 J 接口标志位，用做多播传输
3、 R 复标志位用做多播传输
4、 G RREP 标志位指出是否以多播的形式传输到目的 IP 地址
5、 D 目的地唯一标志，指出只有目的接地那才能回复这个消息
6、 U 未明确序列号，即目的序列号是不明确的
7、 保留：设置为 0，接收时忽略
8、 跳数：从源节点到控制请求节点的跳数。

9、 RREQ ID：和源节点 IP 地址相关联的唯一标识 RREQ 信息的数字
10、 目的 IP 地址：一条路由请求的目的节点的 IP 地址
11、 目的序列号：源节点最新接收到的同相目的节点路由的序列号
12、 源节点 IP 地址：最初发出路由请求的节点的 IP 地址。
13、 源节点序列号：指向发出路由请求的源节点的路由的当前可用序列号。

图 2.1　RREQ 数据包信息格式

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。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，我们称之为反向路由。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（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大时，才认为这条路由是有效的。(目的节点序列号 >RREQ, 更新。)

2.2.2　路由应答报文

RREP 路由应答报文包括以下几个部分

1、 类型：控制报文的类型，RREP 为 2
2、 R 回复标志位，标志多播传输
3、 A 标志位必要的回复信息
4、 保留：设置为 0，接收时忽略。
5、 前缀大小：若非零，与被请求的目的节点具有相同路由前缀的任何节点可能用到的下一跳。
6、 跳数：从源节点到目的节点的跳数，对多播路由，这是指传送 RREP 的多跳树的总跳数。

7、 目的 IP 地址，目的节点的 IP 地址。

8、 目的序列号：和路由关联的序列号

9、 源 IP 地址：最初发出 RREQ 的节点的 IP 地址。

10、 生存周期：节点收到确认路由有效地 RREP 所需时间，单位为 ms

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。

只有在以下情况下才会产生 RREP: 该节点本身就是目的节点；该节点是中间节点，但是他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 更新他们各自指向目的节点的路由信息。节点只对第一次收到的 RREQ 发送 RREP 应答帧。

图 2.2　RREP 数据包信息格式

2.2.3　路由错误报文

路由错误 RERR 报文包括：

1、 类型：控制报文的类型，RERR 为 3

2、 N：不能删除标志，当一个节点在执行本地链路修复时，该设置用来指明上机节点不能删除这条路由

3、 保留：设置为 0，接收时忽略。

4、 目的跳数：在此信息中包括的不可到达的目的地的个数，至少设置 1。

5、 不可达的 IP 地址：由于链路中断造成的不可到达的目的接地那的 IP 地址。

6、 不可达的序列号：通向目的节点的路由表中的序列号，并列在先前不可到达的目的节点的 IP 地址的位置。

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。

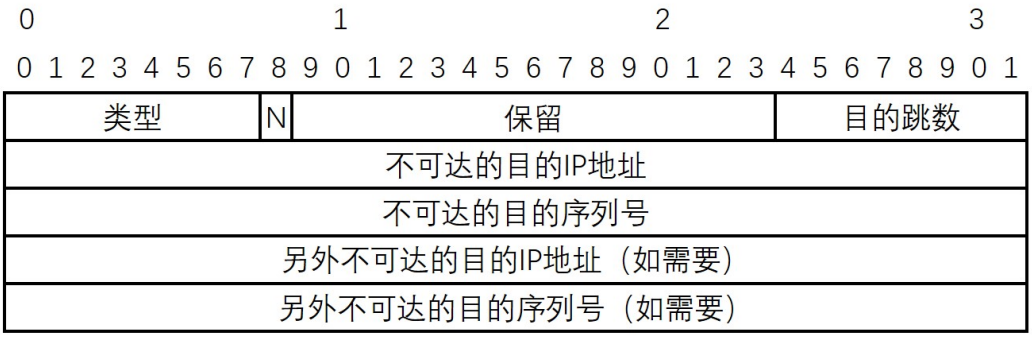

图 2.3　RERR 数据包信息格式

2.2.4　路由应答确认报文

路由应答 RREP-ACK 报文包括

1、 类型：控制报文的类型，RREP-ACK 为 4

2、 保留：设置为 0，接收时忽略。

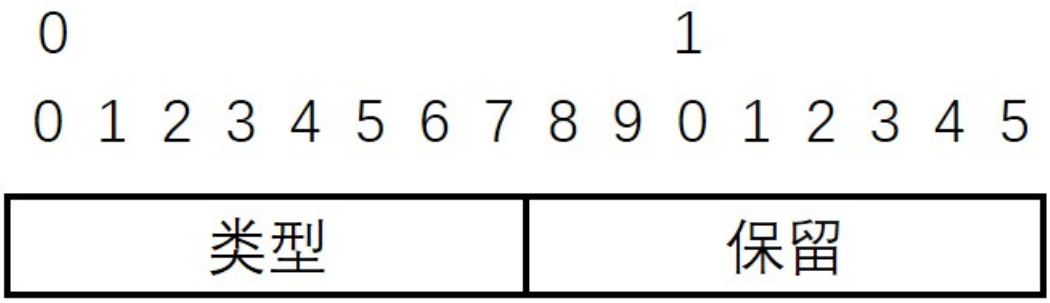

图 2.4　RREP-ACK 数据包信息格式

2.3　路由协议运行

AODV 的运行主要包括路由发现和路由维护两个过程。

2.3.1　路由发现

路由发现过程一般包括：源节点产生路由请求、反向路由的建立、中间节点对路由请求的转发和处理、目的节点产生路由应答，以及对路由应答的转发等。

2.3.1.1　源节点产生路由请求

如果节点需要一条到目的节点的路由而该路由并不可用，则该节点发起一个路由发现过程，广播一个 RREQ。这种情况可能是：目的节点之前对于当前节点是位置的，或者曾经有效的到目的节点的路由已经过期或标记为无效。每个节点都有两个独立的技术

器。节点序列号和广播 ID，RREQ ID 和源节点的 IP 地址联合起来标志一个独一无二的 RREQ 报文，广播出一个 RREQ 以后，源节点等待 RREP 的到来。如果在一个特定的等待时间内没有获得路由，则节点广播另一个 RREQ，视图重新进行路由发现过程，每一次新的尝试都必须增加广播 ID 并更新 RREQ。使用节点序列号可以保证所有的路由信息都是最新的而且无环路。等待路由的用户数据分组应该进行缓存。当路由发现过程达到最大重试此时而仍然没有获得到达目的节点的路由时，则所有要发往对应目的节点的用户数据分组应该从缓存中丢弃，并发送一个目的节点不可达信息给应用层。为了防止网络拥塞，节点在重传 RREQ 的时候，需要使用二进制退避算法。为了防止 RREQ 在不必要的网络范围内传播，源节点要使用拓展环搜索技术。

2.3.1.2 反向路由建立

源节点通常期望与目的节点之间的通信是双向的，所以，路由发现过程不仅使源节点拥有到目的节点的路由，同时也必须使目的节点拥有一条道源节点的反向路由。节点收到 RREQ 后，根据 RREQ 中的相关信息建立或更新到上一跳的反向路由。路由请求消息传播到目的节点的过程会自动建立起一条返回源节点的反向路由。为了建立反向路由，接地那记录最先受到 RREQ 的邻居节点的地址，反向路由的有效时间是至少使得 RREQ 在网络中传播并产生的一个回复信息。

2.3.1.3 处理和转发路由请求

如果受到路由请求信息的节点是中间节点，没有到目的节点的有效路由，而且它受到 RREQ 消息的 IP 报文头部 TTL 值大于 1，它就必须转发路由请求。为了完成对 RREQ 的更新，节点将 IP 报文中的 TTL 字段减去 1，并且将跳数加 1，最后设置目的节点的序列号值。完成对 RREQ 的更新后，节点通过其所有配置接口将 RREQ 广播至地址 255.255.255.255。然后检查在确定时间内，它是否收到具有相同 RREQ ID 和源节点 IP 的 RREQ 报文。如果收到同样的 RREQ，则节点丢弃新收到的 RREQ。

2.3.1.4 产生路由应答

如果一个收到 RREQ 消息的节点具有一条足够新的路由来满足该路由请求，或者它本身就是目的节点，那么该节点就会产生一个 RREP 消息。节点赋值 RREQ 报文中的目的节点 IP 地址和源节点序列号到 RREP 的对应字段。

目的节点产生路由应答：如果产生 RREP 的节点是目的节点本身，假如 RREQ 不保温中的序列号等于目的节点本身的序列号加 1，则节点在产生 RREP 消息之前必须报自己的序列号再加 1；否则，不改变它的序列号，目的节点将其序列号放入 RREP 的目的节点序列号字段当中，并把跳数字段值设置为 0。

中间节点产生路由应答：当产生 RREP 的节点是源节点到目的节点路径上的一个中

间节点时, 它将复制目的节点序列号至 RREP 协议帧中的目的节点列号字段。中间节点将通往目的节点的下一跳节点放入反向路由表项的先驱表中, 用来更新发路由。中间节点将它到目的节跳数放入 RREP 中的跳数字段。

收到 RREQ 并以 RREP 进行响应以后, 节点将丢 RREQ。如果中间节点对每一出的 RREQ 都进行应答, 目的节点就不能收到任何 RREQ 的拷贝, 也就不可能知道到源节点的路由了。为了避免这种情况, 源节点应该在 RREQ 消息中设置“G”标记 (Gratuitous, 免费的), 使得中间节点向源节点返回 RREP 的同寸必须向目的节点单播一个 RREP, RREP。被发送到去往目的节点路径上的下一跳, 就像目的节点已经发出了到源节点的一个 RREQ，而这个 RREP 正是对假想 RREQ 进行应答一样。

一旦建立了 RREP, 当前节点把 RREP 单播至通主源节点的反向路由下一跳。随着 RREP 逐步被转发至源节点, 跳数字段也在每次发送时加 1, 这样, 当 RREP 到达源节点时, 跳数就代表从目的节点到源节点的距离。

2.3.1.5 接收和转发路由应答

当节点收到 RREP 消息, 它首先建立或更新到上一跳的路由, 然后 RREP 中的跳数值加 1, 将新一跳计入其中, 此时跳数为“新跳数”。如果当前节点到达目的节点的转发路由不存在, 则建立转发路由; 否则, 节点对 RREP 消息目的节点的序列号与此前已知目的节点的序列号进行比较。如果此前已知的目的节点的序号无效, 或该序列号有效, 但是 RREP 消息中的较大, 或两者同样大小, 但是 RREP 消息中表明的“新跳数”比此前已知的到达目的节点的跳数小, 那么节点就需要对现存的路由进行更新。

如果当前节点不是发起路由发现过程源节点, 而且已经建立或更新了转发路由, 节点将使用路由表项中的信息向源节点转发 RREP 消息任意节点在发送 RREP 时, 其通往目的节点的先驱表就会更新: 把 RREP 被转发到的下跳节点加入到通往 RREQ 目的节点的路由表项的先驱表中。最后, 到目的节点的下一跳的驱表更新为含有到达源节点的下一跳。

2.3.2 路由维护

路由维护一般包括向邻节点发送 Hello 信息，广播包含不可达的节点信息的 RERR 信息，路由过期以及路由删除。

2.3.2.1 Hello 信息

节点可以通过广播本地 Hello 消息来提供自己的连接性信息。节点只能在自己是一条有效路由的部分是才能使用 Hello 消息。在特定周期内，节点检查最近周期时间内是否以及广播报文（如 RREQ 等），若没有，则会广播一个 TTL 值为 1 的 RREP，称为 Hello 信息。

节点可以通过侦听来自邻居节点的 Hello 信息来确定其连接性。如果节点收到了一个来自邻居节点的 Hello 信息。然后在一个特定时间段内没有收到来自该邻居的任何报文，节点就认为到达该邻居的链路是断开的，并今夕路由出错的处理。只要节点收到来自邻居的 Hello 消息，都应确认自己有一条到达该邻居的有效路由，如果有必要的话，建立一条这样的路由，由 Hello 消息建立，不被任何其他活动路由使用的路由，将具有有空的先驱表。当邻节点移走或这条路由超时无效时，将不触发路由错误报文（RRER）消息。如果路由已经存在，就应该延长这条路由的生存期。

2.3.2.2 RERR 消息

RERR 消息可以广播（有很多先驱），可以是单播（只有一个先驱），或交互式的单播至所有先驱者（广播不合适）。

节点在 3 种情况产生 RERR 消息:

1、 节点检测到一条正在传输数据的路由的下一跳链路断开

2、 节点收到去往某个目的节点的用户数据分组，而节点没有对应有效路由并且没有在进行修复

3、 从邻居节点收到一个关于一条或多条有效路由的 RERR。

路由表中的生存期字段起着双重作用，对于有效路由它是过期时间，对于无效路由它是删除时间。如果一条无效路由收到一个请求使用这条路由的数据分组，它的生存期字段更新为当前时间加上一个特定时间。

2.3.2.3 本地修复

当一条活动路由发生链路中断时，如果中断出的上游节点发现自己到目的节点的跳数小于一个预设特定值，这个节点可能会选择本地修复。为了修复链路中断，节点将关于目的节点的序列号加 1, 然后广播一个寻求得到该目的地的 RREQ。本地修复的尝试对于数据分组的源节点通常是不可见的。

如果在发现周期结束时间内，修复节点没有收到关于目的节点的 RREP（或者其他建立或更新路由的控制消息），则应发送一个关于该目的地节点的 RERR 消息。另一方面，如果节点在路由查找期间收到一个或多个 RREP（或者其他建立更新路由的控制消息），它首先将新路由的跳数与目的节点无效路由表项的跳数值进行比较。如果新确定的到目的地节点的路由的跳数与目的节点无效路由表项的跳数值进行比较。如果新确定的到目的地节点的 RERR 消息，并设置“N”比特标记。发起及诶安东尼收到带有“N”标记的 RERR 消息时，如果这个消息来自沿着去往目的地路由的下一跳节点。那么发起者及诶按可以选择重新发起的一个路由发现过程。

当有效路由的链路中断时，通常有多个目的地的节点变得不可达。中断链路的上一

级节点只视图立即对一个正在进行数据报文的传送目的地节点进行本地修复，其他使用同样里阿奴的路由必须被标记为无效，但是进行本地修复的节点可以将每一条新丢失的路由标记位本地可修复。因此，路由修复是按需进行的，如果没有收到关于某条路由的数据报文，这条路由就不会被修复。另外一种做法是取决于本地拥塞，节点可以不用等待新的数据报文的到达，就开始为其他路由建立本地修复的过程，但是这样可能会带来不必要的开销。

2.4 实例分析

本部分结合具体 AODV 实例对整个过程进行分析。图中 Ao-Hoc 网络中四个节点 S，A，B，C 要依据 AODV 路由协议建立每个节点的路由表，并对该网络进行路由维护，以及路由错误的处理方法。以上所述的协议内容过于抽象，结合具体的实例来看会更加直观。

2.4.1 路由信息格式以及拓扑说明

为了方便讲解，我们把路由信息帧以及路由表简化了。如图 2.5，路由表有“目的 IP 地址，下一跳 IP 地址，跳数，源节点序列号”四个字段。路由请求消息 RREQ 格式简化为“< 目的 IP 地址, 目的序列号,RREQ ID, 源 IP 地址, 源序列号, 跳数 >”，其中 RREQ ID 为路由请求信息的标号，用于避免发送 RREQ 的节点重复发送。路由回复消息 RREP 的格式简化为“< 目的 IP 地址, 目的序列号, 源 IP 地址, 跳数 >”。路由错误信息 RERR 的格式简化为“< 不可达的目的 IP 地址，不可达的目的序列号，跳数 >”。

图 2.5　简化后的路由信息帧及路由表

如图 2.6，节点 S，A，B，C 按图中拓扑排列，首先默认四个节点之间没有建立路

由表项。每个节点都负责维护自己的路由表，图中为实际路由表的简化版，省略了前驱节点链等一些本实例中没有用到的表项。假设，节点之间发生的路由信息的传递不会因链路等问题丢失数据包。

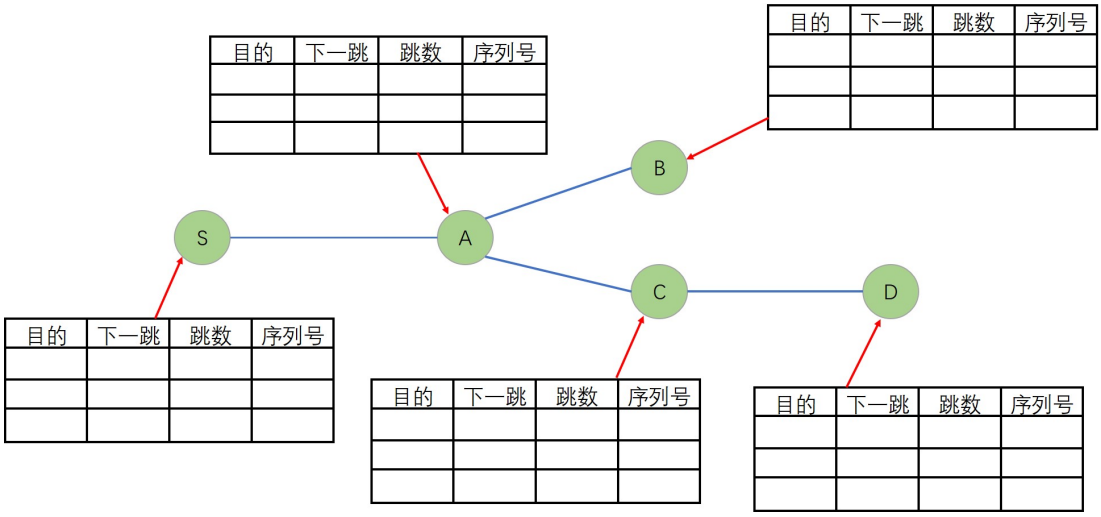

图 2.6　未执行 AODV 协议的 Ao-Hoc 网络拓扑图

2.4.2　路由请求实例

如图 2.7，节点 S 现在要建立一个去节点 D 的路由，首先查找 S 自己的路由表，是否有到 D 的表项，显而易见，没有到 D 的路由表项。所以节点 S 要生成路由请求 RREQ，并向网络中广播该 RREQ。

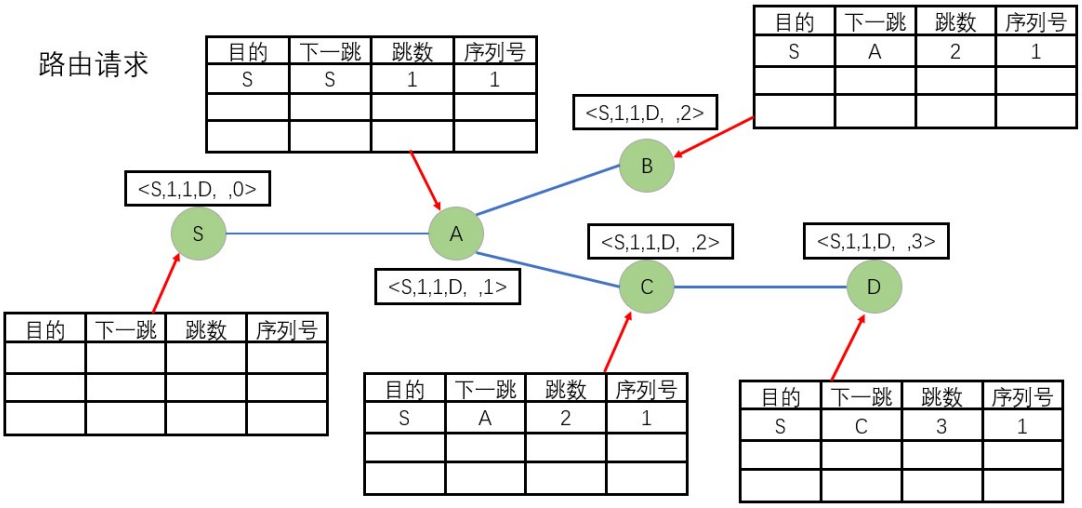

图 2.7　路由请求实例图

节点 A 接收到 S 广播发来的 RREQ，首先把 RREQ 中的信息，即源 IP 地址，源序列

号，跳数添加到自己的路由表中，建立反向路由。再检查是否是发给自己的，显然不是，便查找自己的路由表，是否有到 D 的路由。图中 A 没有与 D 相关的路由，把原 RREQ 中的跳数加 1，便广播修改后的 RREQ。

这时候若 S 又收到了发回来的 RREQ，并不会对其进行处理。原因是 AODV 会控制节点把发送出去的数据包放入缓存区中，接收到的数据包会和该数据包对比，如果首部中的源地址和类型一样，便会丢弃接收到的数据包。

同样的方法，节点 B 和节点 C 接收到 A 广播到的 RREQ 信息，首先建立反向路由，查询自己路由表，没有便跳数加一广播发送改动后的 RREQ。最终，目的地址 D 接收到了该 RREQ 请求。

2.4.3 路由回复实例

如图 2.8，目的节点 D 接收到 RREQ 后，先建立了到节点 S 的反向路由，把 RREQ 中的信息提取，来填充产生路由回复消息 RREP 消息，即源 IP 地址为 S，目的地址为 D，目的序列号为 D 的序列号 120。并沿反向路由发送给 C。

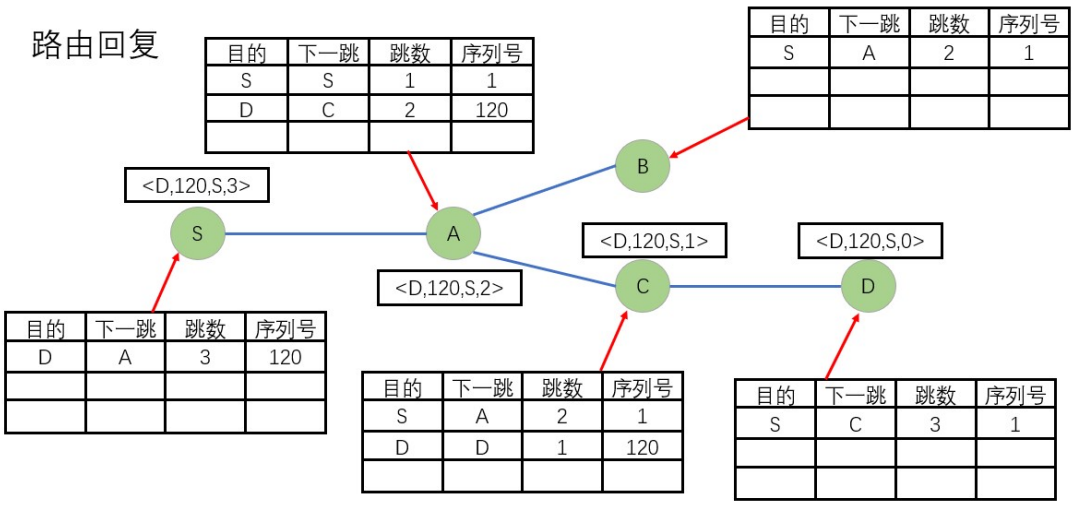

图 2.8　路由回复实例图

节点 C 接收到该 RREP 消息，提取 RREP 中的源地址 IP，并在自己的路由表中查找，找到了带有源地址 IP 也就是 S 的表项，把 RREP 的跳数加 1，沿找到的表项转发给节点 A。A 也是同样的操作转发给了发送 RREQ 请求的源节点 S，S 从 RREP 消息中提取信息填写自己的路由表。

2.4.4 路由维护实例

AODV 拓扑网络中的节点都已经有了自己的路由表项，为了保证网络的连接性，他们需要确保自己所维护的路由是有效的。每隔一段时间，网络中的节点都会向自己的邻居节点发送 TTL 为 1 的 RREP 消息，也就是 Hello 消息来确保路由的连接状态。节点接收到邻居节点发送的 Hello 消息，需要检查自己路由表是否有邻居表项。

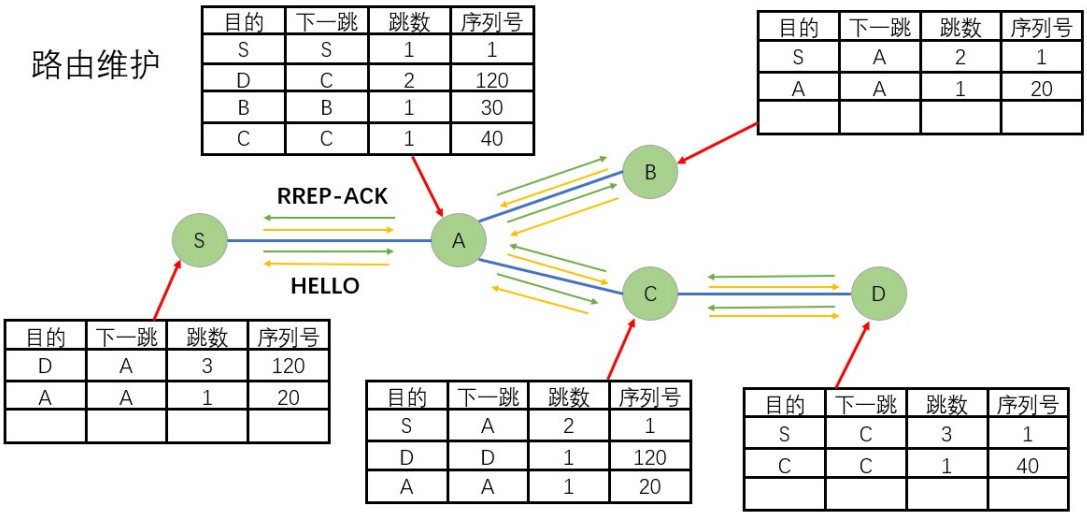

图 2.9　路由维护发送 hello 消息图

如图 2.9，只有从 S 到 A 到 C 再到 D 的路由，而且 A 没有到其邻居节点 B 和 C 的直接路由。所以接收 Hello 的节点在邻居节点不在路由表中时需要建立一个表项，并把 Hello 的内容填充好，发送一个 RREP-ACK 消息，表示接收到了。发送 Hello 信息的节点才会认为该路由有效，并将路由有效时间延长。

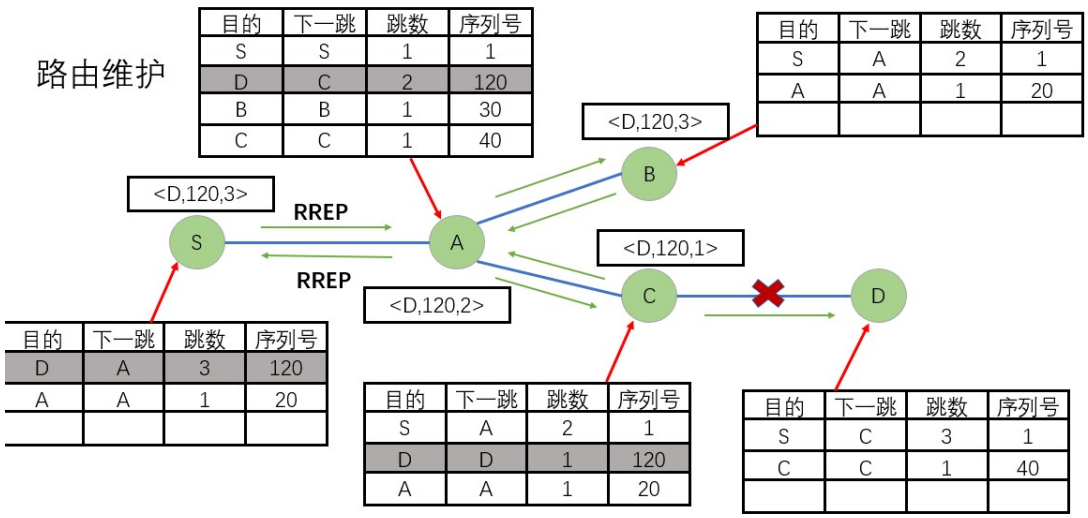

图 2.10　路由错误实例图

如果链路中断导致路由失效，如图 2.10，节点 C 和节点 D 之间路由失效，也就是节点 C 多次向 D 发送 Hello 并没有收到 RREP_ACK 消息，节点 C 认为路由失效，节点 C 把自己路由表中带有 D 的路由置为失效。然后以 C 的 IP 以及序列号创建 RERR 消息。并将其广播至整个网络。接收到 RERR 消息的节点，按照不可达节点 IP 查找路由表项，将查找到的置为无效，并将 RERR 跳数加 1 再广播出去。

这就是一个简单的 AODV 路由协议的路由请求，回复以及维护的实例。在复杂网络拓扑中，要实现 AODV 路由协议，还需要对可以修复的路由进行修复。总之，对于使用 AODV 路由协议的节点来说，很关键的是要维护节点序列号以及路由表。这样整个网络才能避免无穷计算的问题。

3 代码介绍

本项目是基于 aodv-uu-0.9.6 协议的代码分析，为了便于阅读代码，我们给原代码的附上了自己的注释，并对涉及到的 Linux 内核的结构体在网上找到了对应的定义。分析过程中使用的是 Source Insight 软件，目前托管到 github 仓库上 (https://github.com/DeanZhong912/NSA2019-Group10)。

3.1 模块介绍

按照用户层和内核层的区别，把 aodv-uu-0.9.6 的代码按模块基本功能划分为如表 3.1 的 21 个模块。

	模块名称	模块功能	划分
核心层	kaodv_mod	内核主程序, 初始化和注销进程	1
	kaodv_netlink	内核层向用户层进行通信	1
	kaodv_debug	显示核心层处理流程信息	1
	kaodv_ipenc	对内核路由表中 IP 地址编码	1
	kaodv_queue	将数据包放入内核队列	1
	kaodv_expl	内核路由信息到期列表	1
用户层	nl	完成用户层向内核层的通信	1
	adov_socket	接收和发送 aodv 控制信息	2
	aodv_rerr	RERR 消息处理模块	2
	aodv_rreq	RREQ 消息处理模块	2
	aodv_rrep	RREP 消息处理模块	2
	aodv_hello	HELLO 消息处理模块	2
	aodv_neighbo	NEIGHBOR 消息处理模块	2
	endian	选择网络字节顺序	2
	debug	显示 aodvd 线程处理和路由信息	3
	params	定义 aodv 草案中提到的参数	3
	aodv_timeout	对所有 aodv 协议进行超时操作	3
	timer_queue	定时器队列	3
	routing_table	AODV 路由表项存储	3
	seek_list	记录目的节点的 IP 地址序列	3
	locality	查找主机模块	3

表 3.1 代码功能模块

在 aodv-uu 路由协议实现的设计中，路由体系结构分为路由功能模块和转发功能模块对数据包的处理分别在用户层和核心层进行。

路由功能模块作为后台进程在用户层运行，主要完成与其他网络节点的信息交流，采用适当的路由算法建立路由、更新和维护 Linux 内核路由表。转发功能在操作系统内核中实现，该模块根据路由表中的信息，将需要发送的网络数据分组通过正确的网络接口发送到下一跳节点。

按照代码主要的工作内容，将这 21 个模块划分为三个主要部分，分别是内核层 AODV 协议及与用户层交互部分，用户层 AODV 协议过程部分和用户层路由表与超时维护等部分。在上表中分别对应 1，2，3。

3.2 全局变量

首先介绍全局变量，这里的全局变量是 main 函数的全局变量，其中的有一些在协议中的其他函数被调用。

变量名	数据类型	详细说明	备注
log_to_file	int	记录调试输出	Debug.c 中调用
rt_log_interval	int	路由表日志间隙	Debug.c 中调用
unidir_hack	int	检测和避免单向链接	hello.c 中调用
rreq_gratuitous	int	强制在所有 RREQ 上设置免费标志	rreq.c 中调用
expanding_ring_search	int	禁用扩展环搜索 RREQs	rreq.c 和 timeout.c 中调用
internet_gw_mode	int	启用实验性的 Internet 网关支持	rreq.c 和 nl.c 中调用
local_repair	int	标志本地修复	timeout.c 中调用
receive_n_hellos	int	从主机处接收到的 hello 消息数	hello.c 中调用
hello_jittering	int	切换 hello jittering	hello.c 中调用
optimized_hellos	int	仅在转发数据时发送 hello	hello.c 和 rrep.c 中调用
ratelimit	int	限制 rreq 和 rerr 消息发送速率	socket.c 中调用
progname	char*	程序名	程序设置选项时调用
wait_on_reboot	int	禁用 15 秒等待重新启动延迟	socket.c 和 nl.c 中调用
qual_threshold	int	为控制包设置最小信号质量阈值	nl.c 中调用
llfeedback	int	启用链路层反馈	多个 c 文件中调用
gw_prefix	int	实验性的 Internet 网关支持	locality.c 中调用
worb_timer	struct timer	等待重启计时器	nl.c 中调用

表 3.2 main 函数中全局变量

全局变量由上表所示, 根据使用者对运行 aodv 协议的要求, 在程序运行时设置选项对每一次运行进行自定义设定。选项的用法由 main.c 文件中的 usage 函数 (Line 97-129)

显示。

```
void usage(int status){ //返回帮助
   if (status != 0) {
      fprintf(stderr, "Try '%s --help' for more information.\n", progname);
      exit(status);
   }
   printf("Usage: %s [-dghjlouwxLDRV] [-i if0,if1,..] [-r N] [-n N] [-q THR]\n"
   "-d, --daemon            Daemon mode, i.e. detach from the console.\n"
   "-g, --force-gratuitous  Force the gratuitous flag to be set on all RREQ's.\n"
   "-h, --help              This information.\n"
   "-i, --interface         Network interfaces to attach to. Defaults to first\n"
   "                        wireless interface.\n"
   "-j, --hello-jitter      Toggle hello jittering (default ON).\n"
   "-l, --log               Log debug output to %s.\n"
   "-o, --opt-hellos        Send HELLOs only when forwarding data (experimental).\n"
   "-r, --log-rt-table      Log routing table to %s every N secs.\n"
   "-n, --n-hellos          Receive N hellos from host before treating as neighbor.\n"
   "-u, --unidir-hack       Detect and avoid unidirectional links (experimental).\n"
   "-w, --gateway-mode      Enable experimental Internet gateway support.\n"
   "-x, --no-expanding-ring Disable expanding ring search for RREQs.\n"
   "-D, --no-worb           Disable 15 seconds wait on reboot delay.\n"
   "-L, --local-repair      Enable local repair.\n"
   "-f, --llfeedback        Enable link layer feedback.\n"
   "-R, --rate-limit        Toggle rate limiting of RREQs and RERRs (default ON).\n"
   "-q, --quality-threshold Set a minimum signal quality threshold for control packets.\n"
   "-V, --version           Show version.\n"
   "Erik Nordstr m, <erik.nordstrom@it.uu.se>",progname, AODV_LOG_PATH, AODV_RT_LOG_PATH);
   exit(status);
}
```

3.3 数据结构

在具体代码分析之前，除了全局变量，还有重要的数据结构需要了解。

3.3.1 设备和主机信息结构体

如下代码 (defs.h Line 87-109) 是设备信息结构体和主机信息结构体。主机结构体用于 aodv 节点维护的自己的信息，设备信息是指与本节点相连的接口信息。

```
/* Data for a network device */
struct dev_info {   //设备信息        设备即接口 interface
   int enabled;  /* 1 if struct is used, else 0 有效为1 无效为0*/
   int sock;   /* AODV socket associated with this device AODV与这个设备关联的socket*/
#ifdef CONFIG_GATEWAY
   int psock;   /* Socket to send buffered data packets. 发送缓冲数据包的socket*/
#endif
   unsigned int ifindex; //接口号
   char ifname[IFNAMSIZ];
   struct in_addr ipaddr; /* The local IP address 本地IP地址*/
   struct in_addr netmask; /* The netmask we use 子网掩码*/
```

```
    struct in_addr broadcast; //广播地址
};

struct host_info {
    u_int32_t seqno;  /* Sequence number 序列号*/
    struct timeval bcast_time; /* The time of the last broadcast msg sent 最新一次广播消息发送的时间*/
    struct timeval fwd_time; /* The time a data packet was last forwarded 最新一次转发数据包的时间*/
    u_int32_t rreq_id;  /* RREQ id */
    int nif;    /* Number of interfaces to broadcast on 需要广播的接口的数量*/
        struct dev_info devs[MAX_NR_INTERFACES+1]; /* Add +1 for returning as "error" in
            ifindex2devindex.
                                        10+1 多加1是为了返回 作为一个错误在下面的那个函数中*/
};
```

3.3.2 路由表结构体

3.3.3 RREQ 结构体

3.3.4 Socket 结构体

4 具体代码分析

本部分按照内核层 AODV 协议及与用户层交互部分，用户层 AODV 协议过程部分和用户层路由表与超时维护等的顺序进行具体的代码分析。

4.1 内核层 AODV 协议及与用户层交互

内核层 AODV 协议及与用户层交互部分由仲点负责分析，这一部分从内核关键变量和结构体，Netfilter 处理数据包功能，以及与用户层的交互三个部分选取关键代码进行分析。

4.1.1 内核关键变量及结构体

我们总结了内核中的重要变量并将他们汇总到表格中。

变量名	数据类型	详细说明	备注
expl_lock	rwlock_t	对内核 expl 表进行读写锁控制	在 kaodv_expl.c 中
queue_lock	rwlock_t	对内核 queue 队列进行读写控制	在 kaodv_queue.c 中
ifilock	rwlock_t	对接口信息链表进行读写控制	在 kaodv_mod.c 中
hooknum	int	控制数据包在内核的流动	在 kaodv_mod.c 中
is_gateway	int	判断是否为网关	作为函数参数使用
active_route_timeout	int	有效路由超时时间	设定为 3000
qual_th	int	信号质量阈值	判断数据包是否失效
pkts_dropped	int	丢弃的数据包数量	反映链路状态
skb	struct *sk_buff	指向 sk_buff 的指针	内核中的关键结构
aodvnl	struct nlsock	用户层向核心层通信的协议套接字	用于用户态到内核态的协议
rtnl	struct nlsock	用户层向核心层通信的路由表套接字	用于用户态到内核态的路由表
kaodvnl	struct sock*	核心层向用户层通信的套接字	属于 Netlink 模块

表 4.1 内核部分全局变量

谈及内核，就要涉及到读写操作，aodv-uu 定理了两个读写锁 expl_lock 和 queue_lock，分别对内核中的路由信息超时表和路由信息队列两个结构进行读写控制。expl 的结构体见如下代码 (kaodv_expl.h Line 29-36)

```
struct expl_entry {          //路由表信息到期列表
    struct list_head l;      //expl表头
    unsigned long expires;   //到期时间
    unsigned short flags;    //是否可修复
    __u32 daddr;             //__u32这个变量占4字节      目的地址
    __u32 nhop;              //下一跳
```

```
    int ifindex;//编号
};
```

可见 expl 的结构就是一个双向的链表，有到期时间，目的地址，下一跳，是否可修复四个表项。同样在 kaodv_expl.h 中声明了七个函数 (初始化，清空，添加，删除，更新等) 并在相应的 c 文件中进行了实现。该部分位于内核部分，根据操作系统的知识，对 expl 表进行读写操作要使用 expl_lock 进行控制。

接口信息表结构体 (kaodv_mod.h Line 65-70) 代码如下：

```
struct if_info {
    struct list_head l;//填表首部
    struct in_addr if_addr;//接口地址
    struct in_addr bc_addr;//广播地址
    struct net_device *dev;//接口连接上的设备
};
```

在同一文件里还定义了对接口信息表的，增加，清空以及根据 index 读取广播地址以及接口 IP 地址。

queue 结构体以及相关的定义 (kaodv_queue.h Line 51-62)

```
struct kaodv_queue_entry {   //队列表,本质上就是一个双向链表
    struct list_head list;   //表头
    struct sk_buff *skb;     //缓冲区buff结构 sk_buff
    int (*okfn) (struct sk_buff *);
    struct kaodv_rt_info rt_info; //内存中的路由信息
};
struct kaodv_rt_info {
    __u8 tos;                //两字节，服务类型 typeofservice
    __u32 daddr;             //目的地址
    __u32 saddr;             //原地址
};
```

queue 是一个由 sk_buff 以及 rt_info 元素组成的双向链表，sk_buff 结构体是用来用来封装网络数据，aodv-uu 协议内核部分对数据的处理是以 sk_buff 结构为单元进行的。

在 kaodv_queue.c 文件中定义了对内核中的信息队列 queue 的增、删、该、查、清空以及初始化的操作。

对 queue , if_info 以及 expl 的改动是在 kaodv_mod.c 文件中的 hook 函数进行的，关于这一部分的介绍放在 Netfilter 处理数据包部分介绍。

既然内核使用了 sk_buff 作为数据处理的单元，那就有必要介绍一下 sk_buff 结构。sk_buff 是定义在 <linux/skbuff.h> 中，它的基本代码如下：

```
struct sk_buff {
    struct sk_buff  * volatile next;
    struct sk_buff  * volatile prev;//构成队列
```

```
    struct sk_buff  * volatile link3; //构成数据包重发队列
    struct sock   *sk; //数据包所属的套接字
    struct device   *dev; //接收该数据包的接口设备
    struct sk_buff  *mem_addr; //该sk_buff在内存中的基地址，用于释放该sk_buff结构
    union {          //联合类型，表示数据报在不同处理层次上所到达的处理位置
       struct tcphdr *th; //传输层tcp，指向首部第一个字节位置
       struct ethhdr *eth; //链路层上，指向以太网首部第一个字节位置
       struct iphdr *iph; //网络层上，指向ip首部第一个字节位置
       struct udphdr *uh; //传输层udp协议，
       unsigned char *raw; //随层次变化而变化，链路层=eth，网络层=iph
       unsigned long seq; //针对tcp协议的待发送数据包而言，表示该数据包的ACK值
    } h;
    struct iphdr  *ip_hdr; //指向ip首部的指针  /* For IPPROTO_RAW */
    unsigned long   mem_len; //表示sk_buff结构大小加上数据部分的总长度
    unsigned long   len; //只表示数据部分长度，len = mem_len - sizeof(sk_buff)
    unsigned long   fraglen; //分片数据包个数
    struct sk_buff  *fraglist; /* Fragment list */
    unsigned long   saddr; //源端ip地址
    unsigned long   daddr; //目的端ip地址
    unsigned long   raddr; //数据包下一站ip地址  /* next hop addr */
    unsigned char   data[0]; //指向该层数据部分
    ...
};
```

根据 sk_buff 的代码我们画出了 sk_buff 结构体的简单示意图 3.1。图中 sk_buff 主要部分由四条个指针组成。head 是指向分配给的线性数据内存首地址 data 指向的数据负载首地址，在各个层对应不同的数据部分，tail 指向数据的结尾，end 指向分配的内存块的结尾。从侧面看出 sk_buff 结构基本上是贯穿整个网络栈的非常重要的一个数据结构。

我们不太过具体 sk_buff 结构，分析协议的时候注意它是类似课上讲过的 mbuff 结构，但要比 mbuff 更加复杂且应用广泛。为了分析的方便，将它单独列出来。aodv 的相关数据存放在数据存储区，所以就不介绍分片结构体了。

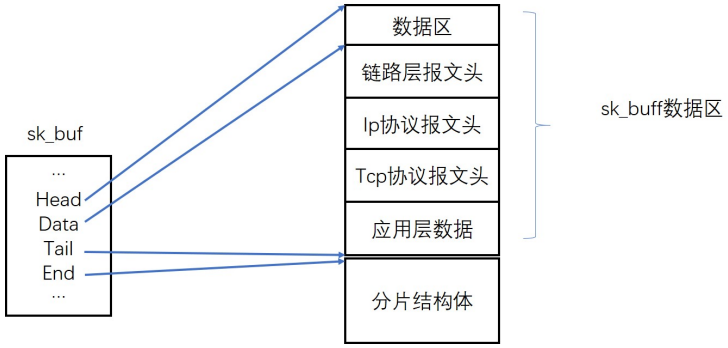

图 4.1　sk_buff 结构体示意图

内核部分还有套接字的结构需要注意。nlsock 结构体是实现到内核的 Netlink 套接字通信通道。传递路由信息和刷新消息。

```
struct sockaddr_nl{
    sa_family_t nl_family;
    unsigned short nl_pad;
    __u32       nl_pid; //进程pid
    __u32       nl_groups;//多播组掩码
};
struct nlsock {
    int sock;
    int seq;
    struct sockaddr_nl local;//sockaddr_nl结构 AF_NETLINK nl_pad 进程pid 多播组掩码
};
```

4.1.2 Netfilter 处理数据包

Netfilter 在内核层实现 aodv 起到了关键性的作用。Netfilter 是由处于 Linux 协议栈中不同点上的五个钩子 (hook) 函数组成，根据数据包 hooknum 值的不同进行对应不同的操作。

如下的代码定义了 Netfilter 五个 hook 的宏 (kaodv_mod.c Line 55-60)，hooknum 的取值就是从这里面取。

```
#define NF_INET_PRE_ROUTING NF_IP_PRE_ROUTING //进入路由代码之前
#define NF_INET_LOCAL_IN NF_IP_LOCAL_IN //发往本机的数据报，经过该宏的处理传入上层
#define NF_INET_FORWARD NF_IP_FORWARD //应该被转发的数据报，由该宏进行处理
#define NF_INET_LOCAL_OUT NF_IP_LOCAL_OUT//本地产生的数据报，由该宏进行处理
#define NF_INET_POST_ROUTING NF_IP_POST_ROUTING//应该转发的数据报经过该宏处理后发往网络
#define NF_INET_NUMHOOKS NF_IP_NUMHOOKS
```

hook 函数的每一种 hook 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:

1、 NF_ACCEPT: 继续正常的数据包处理;
2、 NF_DROP: 将数据包丢弃;
3、 NF_STOLEN: 由 Hook 函数处理了该数据包，不要再继续传送;
4、 NF_QUEUE: 将数据包入队，通常交由用户程序处理;
5、 NF_REPEAT: 再次调用该 Hook 函数。

内核 aodv 程序通过 Netfilter 实现了对数据分组的过滤和修改等功能。基本过程如下:

如图 3.2，数据包网络入口以后，首先对 IP 进行校验后，经过 hook 函数的 NF_-IP_PIFORWARD 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 hook 函数的 NF_IP_LOCAL_IN 传递给上层协议，若该数据包应该被转发则把它被交给 NF_ IP_PIFORWARD 处理，转发的数据包经过 hook 函数 NF_IP_LOCAL_OUT 处理后，再发送到网络上。本地产生

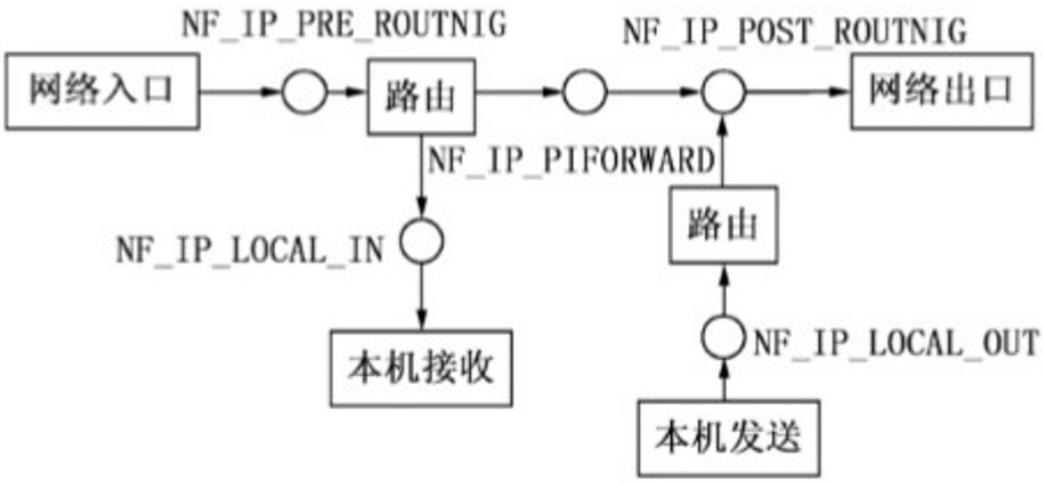

图 4.2　Netfilter 过滤数据分组示意图

的数据包经过 hook 函数 NF_IP_LOCAL_OUT 处理后，进行路由选择处理，然后经过 NF_IP_POST_ROUTNIG 处理，最终发送到网络上。

aodv 在内核层定义了回调函数并与 Netfilter 的其中三个 hook 函数，流入数据包在决策路由前执行的函数 NF_IP_PRE_ROUTING、本地产生数据包流出路由前执行的函数 NF_IP_LOCAL_OUT 、在本地数据包或转发数据包在发送之前的函数 NF_IP_POST_-ROUTING 进行了挂载 (kaodv-mod.c Line 193-287) 因为 kaodv_hook 函数是内核部分的核心函数，需要着重地进行分析。

Line 194-227 回调函数挂载 NF_INET_PRE_ROUTING 本机接收数据包

```
  switch (hooknum) {//检验hooknum
case NF_INET_PRE_ROUTING:
  kaodv_update_route_timeouts(hooknum, in, iph);
  /* If we are a gateway maybe we need to decapsulate? */
  if (is_gateway && iph->protocol == IPPROTO_MIPE &&
      iph->daddr == ifaddr.s_addr) {//如果是网关且ip协议为55，并且数据包的目的地址是自己
    ip_pkt_decapsulate(skb);//解封装并转发
    iph = SKB_NETWORK_HDR_IPH(skb);//获取skb到iph里
    return NF_ACCEPT;
  }
  /* Ignore packets generated locally or that are for this
   * node. */
  if (iph->saddr == ifaddr.s_addr ||
      iph->daddr == ifaddr.s_addr) {//如果是自身产生的数据包则忽略
    return NF_ACCEPT;
  }
  /* Check for unsolicited data packets */
  else if (!kaodv_expl_get(iph->daddr, &e)) {//检查是否是未请求的，expl表中没有的
    kaodv_netlink_send_rerr_msg(PKT_INBOUND, iph->saddr,
```

```
            iph->daddr, in->ifindex);//向用户层发送路由错误信息，并丢弃该数据包
    return NF_DROP;
  }
  /* Check if we should repair the route */
  else if (e.flags & KAODV_RT_REPAIR) {//检查是否可修复

    kaodv_netlink_send_rt_msg(KAODVM_REPAIR, iph->saddr,
          iph->daddr);//发送路由修复的消息

    kaodv_queue_enqueue_packet(skb, okfn);//把skb进入队列中
    return NF_STOLEN;//交由Hook函数处理，不再继续传送
  }
     break;
```

Line 195 更新内核中的路由表，调用 kaodv_update_route_timeouts 函数，该函数传入的参数 hooknum 为 NF_INET_PRE_ROUTING，知道了该数据包刚到达本机。便使用调用 kaodv_netlink_send_rt_update_msg 函数向用户层发送路由更新消息。kaodv_update_-route_timeouts 调用 kaodv_expl 模块根据数据包的目的地址更新 expl 表项。

Line198-227 根据不同情况对数据包进行处理，如果本机是网关节点，则调用 ip_-pkt_decapsulate 函数对数据包进行解封装以及转发操作。如果目的地或者发送方是本机，回复正常接收。如果是未请求且本机 expl 表中没有的，则创建路由错误信息并发送至用户层，丢弃该数据包。如果是应该修复的路径，则调用相关函数创建路由修复信息，该数据包入队，不继续传送该数据包。

Line 228-284 回调函数挂载 NF_INET_LOCAL_OUT 本机发送数据包

```
 case NF_INET_LOCAL_OUT://本地产生的数据包
  if (!kaodv_expl_get(iph->daddr, &e) ||
     (e.flags & KAODV_RT_REPAIR)) {//如果未找到且该表项可修复

    if (!kaodv_queue_find(iph->daddr))//如果该表项在待处理队列中没有
      kaodv_netlink_send_rt_msg(KAODVM_ROUTE_REQ,
            0,
            iph->daddr);//控制netlink模块发送一遍路由请求

    kaodv_queue_enqueue_packet(skb, okfn);//将它加入队列
    return NF_STOLEN;//交由Hook函数处理，不再继续传送
  } else if (e.flags & KAODV_RT_GW_ENCAP) {
#ifdef ENABLE_DISABLED
    /* Make sure the maximum segment size (MSM) is
       reduced to account for the
       encapsulation. This is probably not the
       nicest way to do it. It works sometimes,
       but may freeze due to some locking issue
       that needs to be fix... */
    if (iph->protocol == IPPROTO_TCP) {
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24))
      if ((*skb)->sk) {
```

```
        struct tcp_sock *tp = tcp_sk((*skb)->sk);
        if (tp->mss_cache > 1452) {
          tp->rx_opt.user_mss = 1452;
          tp->rx_opt.mss_clamp = 1452;
          tcp_sync_mss((*skb)->sk, 1452);
        }
       }
#else
       if (skb->sk) {
        struct tcp_sock *tp = tcp_sk(skb->sk);
        if (tp->mss_cache > 1452) {
          tp->rx_opt.user_mss = 1452;
          tp->rx_opt.mss_clamp = 1452;
          tcp_sync_mss(skb->sk, 1452);
        }
       }
#endif
     }
#endif /* ENABLE_DISABLED */
     /* Make sure that also the virtual Internet
      * dest entry is refreshed */
     kaodv_update_route_timeouts(hooknum, out, iph);//更新路由信息

     skb = ip_pkt_encapsulate(skb, e.nhop);//数据包解封装

     if (!skb)
       return NF_STOLEN;

     ip_route_me_harder(skb, RTN_LOCAL);
   }
       break;
```

Line 230-242 如果 expl 表中没有找到目的节点或者路由的状态是该表项可恢复，便控制 netlink 模块创建路由更新信息并发送给用户层，并将该数据包入队，不继续传送该数据包。

Line 242-284 如果 expl 表中有目的节点，则 kaodv_update_route_timeouts 会根据 hooknum 调用 netlink 模块创建路由信息发送到用户态并更新 expl 表，最后 ip_pkt_-encapsulate 函数对数据包封装，发送到下一跳。

Line 285-287 回调函数挂载 NF_INET_POST_ROUTING 数据包在网络入口

```
 case NF_INET_POST_ROUTING://应该是转发出去的
   kaodv_update_route_timeouts(hooknum, out, iph);//转发出去
 }
```

调用 kaodv_update_route_timeouts 函数，控制 netlink 模块创建路由更新信息并发送给用户层，更新 expl 表。

kaodv_hook 函数最后向 Netfilter 核心代码回复正常处理。

4.1.3 与用户层交互

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 aodvd 后台进程核心路由表的使用状况，aodvd 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。

在核心层，控制用户层 API 的是 kaodv-netlink 模块。该模块使用的套接字是 kaodvnl，创建改套接字的是 netlink_kernel_create 函数。

kaodv-netlink 模块说明之前提到的被 kaodv_mod 模块调用的 kaodv_netlink_build_-msg 函数 (Line 55-89) 和 kaodv_netlink_send_rt_msg 函数 (Line 105-124)。

kaodv_netlink_build_msg 函数的主要作用是根据所传参数，定义传向用户态的 Message 并为其申请内存空间，并填充数据结构为 sk_buff 的 Message。

```
static struct sk_buff *kaodv_netlink_build_msg(int type, void *data, int len)//创建路由信息{
    unsigned char *old_tail;
    size_t size = 0;
    struct sk_buff *skb;
    struct nlmsghdr *nlh;// nlmsghdr结构
    void *m;
    size = NLMSG_SPACE(len);//LMSG_SPACE返回不小于NLMSG_LENGTH(len)且字节对齐的最小数值，它也用于分配消息
        缓存。
    skb = alloc_skb(size, GFP_ATOMIC);//为skbuff申请内核空间,GFP_ATOMIC用来从其他代码中分配内存
    if (!skb)
        goto nlmsg_failure;
    old_tail = SKB_TAIL_PTR(skb);//old_tail指向skbuff的末尾
    nlh = NLMSG_PUT(skb, 0, 0, type, size - sizeof(*nlh));//nlmsg_put把nlh填入skb
    m = NLMSG_DATA(nlh);//nlmsg_data用于取得消息的数据部分的首地址，设置和读取消息数据部分时需要使用该宏。
    memcpy(m, data, len);
    nlh->nlmsg_len = SKB_TAIL_PTR(skb) - old_tail;
    NETLINK_CB(skb).pid = 0; /* from kernel */
    return skb;
    nlmsg_failure:
    if (skb)
        kfree_skb(skb);
    printk(KERN_ERR "kaodv: error creating rt timeout message\n");
    return NULL;
}
```

kaodv_netlink_send_rt_msg 函数的作用是调用 kaodv_netlink_build_msg 函数创建路

由类型的 Message(Line 115)，并根据所传参数填充 Message 中的源 IP 地址和目的 IP 地址 (Line 112-113)，调用 kaodvnl 套接字 (Line 123) 将 Message 发送到用户态。该模块的 kaodv_netlink_send_debug_msg 函数 (Line 91-103)、kaodv_netlink_send_rt_update_msg 函数 (Line 126-148) 和 kaodv_netlink_send_rerr_msg 函数 (Line 150-171) 都是使用的相似的结构，创建了不同类型的 Message 并发送至用户态。

该部分的 kaodv_netlink_receive_peer 函数 (Line 173-231) 用于 kaodvnl 套接字的创建。并不涉及与用户态的交互。

在用户层，控制核心层接口的是 nl 模块。这一模块以将添加路由信息发送到内核的 nl_send_add_route_msg 函数 (Line 534-573) 为例，来说明用户态具体怎么发送信息到内核态。

nl_send_add_route_msg 函数被 rt_table 模块中的 rt_table_insert 函数和 rt_table_update 函数调用，用来给内核态发送消息用以更新内核态中的 expl 表。

```
int nl_send_add_route_msg(struct in_addr dest, struct in_addr next_hop,
int metric, u_int32_t lifetime, int rt_flags,
int ifindex)
{
struct {
struct nlmsghdr n;
struct kaodv_rt_msg m;
} areq;
DEBUG(LOG_DEBUG, 0, "ADD/UPDATE: %s:%s ifindex=%d",
ip_to_str(dest), ip_to_str(next_hop), ifindex);
memset(&areq, 0, sizeof(areq));
areq.n.nlmsg_len = NLMSG_LENGTH(sizeof(struct kaodv_rt_msg));
areq.n.nlmsg_type = KAODVM_ADDROUTE;//类型为添加路由条目
areq.n.nlmsg_flags = NLM_F_REQUEST;
areq.m.dst = dest.s_addr;
areq.m.nhop = next_hop.s_addr;
areq.m.time = lifetime;
areq.m.ifindex = ifindex;
if (rt_flags & RT_INET_DEST) {
   areq.m.flags |= KAODV_RT_GW_ENCAP;
}
if (rt_flags & RT_REPAIR)
   areq.m.flags |= KAODV_RT_REPAIR;
if (nl_send(&aodvnl, &areq.n) < 0) {
   DEBUG(LOG_DEBUG, 0, "Failed to send netlink message");
return -1;
}
#ifdef DEBUG_NETLINK
DEBUG(LOG_DEBUG, 0, "Sending add route");
#endif
return nl_kern_route(RTM_NEWROUTE, NLM_F_CREATE,
AF_INET, ifindex, &dest, &next_hop, NULL, metric);//调用nl_kern_route在内核中创建新表项
}
```

Line 538-555 该函数首先定义存放要发送到内核的结构体变量 areq, 并按照参数对其进行赋值，

Line 564-567 调用 nl_send 函数使用 aodvnl 发送至内核层的 netlink 接口，

Line 571-572 最后调用 nl_kern_route 函数创建修改内核路由信息的 Message，并使用 rtnl 套接字发送过去。如果失败则会返回 Debug 消息。

nl 模块函数的其他函数比如发送无路径信息 nl_send_no_route_found_msg(函数 Line 575-594), 发送删除路由函数 nl_send_del_route_msg 函数 (Line 596-626), 发送配置信息 nl_send_conf_msg(Line 628-649)，都是定义发送数据的信息格式，申请空间并按照参数填充，调用 nl_send 函数使用 aodvnl 套接字与内核态进行了交互，但是三个中只有 nl_send_del_route_msg 函数使用了 nl_kern_route 函数。

该模块的 nl_send(Line 456-481) 函数和 nl_kern_route(Line 486-532) 函数也需要进行分析

nl_send 主要作用是, 把传进来的类型为 struct nlmsghdr 的信息进行一些必要的填充，相当于给发送给内核的消息打上一个 aodv 的身份标签，并将其发送到 netlink 接口以便与内核通信。

```
int nl_send(struct nlsock *nl, struct nlmsghdr *n)
{
    int res;
    struct iovec iov = { (void *) n, n->nlmsg_len };
    struct msghdr msg =
        { (void *) &peer, sizeof(peer), &iov, 1, NULL, 0, 0 };
    // int flags = 0;
    if (!nl)
        return -1;
    n->nlmsg_seq = ++nl->seq;
    n->nlmsg_pid = nl->local.nl_pid;
    /* Request an acknowledgement by setting NLM_F_ACK */
    n->nlmsg_flags |= NLM_F_ACK;
    /* Send message to netlink interface. */
    res = sendmsg(nl->sock, &msg, 0);//把msg发送到netlink接口
    if (res < 0) {
        fprintf(stderr, "error: %s\n", strerror(errno));
        return -1;
    }
    return 0;
}
```

在 nl_kern_route 中，定义了 req 结构体，并为其申请了空间，按照参数对 req 进行填充。调用的 addattr 函数是对 req.nlh 的数据进行填充，最后调用 nl_send, 使用 rtnl 套接

字实现对内核路由表交互信息的发送。

```
int nl_kern_route(int action, int flags, int family,
  int index, struct in_addr *dst, struct in_addr *gw,
  struct in_addr *nm, int metric)
{
    struct {
        struct nlmsghdr nlh;
        struct rtmsg rtm;
        char attrbuf[1024];
    } req;
    if (!dst || !gw)
        return -1;
    req.nlh.nlmsg_len = NLMSG_LENGTH(sizeof(struct rtmsg));
    req.nlh.nlmsg_type = action;
    req.nlh.nlmsg_flags = NLM_F_REQUEST | flags;
    req.nlh.nlmsg_pid = 0;
    req.rtm.rtm_family = family;
    if (!nm)
        req.rtm.rtm_dst_len = sizeof(struct in_addr) * 8;
    else
        req.rtm.rtm_dst_len = prefix_length(AF_INET, nm);
    req.rtm.rtm_src_len = 0;
    req.rtm.rtm_tos = 0;
    req.rtm.rtm_table = RT_TABLE_MAIN;
    req.rtm.rtm_protocol = 100;
    req.rtm.rtm_scope = RT_SCOPE_LINK;
    req.rtm.rtm_type = RTN_UNICAST;
    req.rtm.rtm_flags = 0;
    addattr(&req.nlh, RTA_DST, dst, sizeof(struct in_addr));
    if (memcmp(dst, gw, sizeof(struct in_addr)) != 0) {
        req.rtm.rtm_scope = RT_SCOPE_UNIVERSE;
        addattr(&req.nlh, RTA_GATEWAY, gw, sizeof(struct in_addr));
    }
    if (index > 0)
        addattr(&req.nlh, RTA_OIF, &index, sizeof(index));
    addattr(&req.nlh, RTA_PRIORITY, &metric, sizeof(metric));//实际上也是定义结构，申请空间，并赋值，并没
        有具体操作，具体操作在内核层实现，这里调用rtnl套接字交给了内核处理
    return nl_send(&rtnl, &req.nlh);//调用rtnl套接字修改内核中的路由表
}
```

该模块的 nl_kaodv_callback 函数 (Line 206-353) 和 nl_rt_callback(Line 354-414) 函数用于 nl 模块的初始化，这里不过多介绍。

4.2 用户层 AODV 协议过程

4.3 用户层路由表与超时维护等

4.4 系统框架

综合代码分析的三部分，可以得到 AODV-UU 协议的系统框架图 (图 3.3), 这一部分结合框图介绍我们分析的整个代码的工作流程。

4.4.1 用户层工作流程

在用户层 aodvd 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。

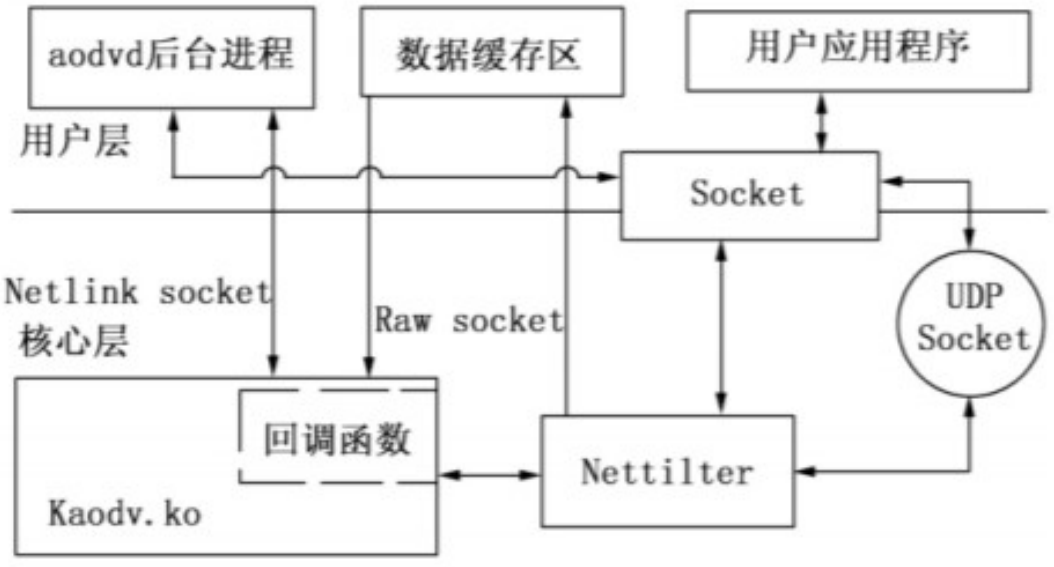

图 4.3　AODV-UU 协议框架结构图

4.4.2 内核层与用户层交互

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 套接字 kaodvnl 将核心路由表 expl 的使用状态传递给用户层，告知 aodvd 后台进程 expl 的使用状况，aodvd 后台进程据此更新路由缓冲表的定时器，同时通过 Netlink 套接字 rtnl 删除内核路由表中过时的路由条目或增加新的路由表项。

4.4.3 内核层工作流程

在本地直接发送或者转发数据包时，首先判断 expl 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) ，也就是调用 netlink_broadcast 函数，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。

4.4.4 路由表及超时管理

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 (Raw Socket) 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，则这些缓存的数据据报文会被丢弃，并通知源节点出错。

5 总结

我们本次做的是对 AODV-uu-0.9.6 路由协议的代码分析,。。。。。