**网络协议栈分析与设计课程大作业**

|  |
| --- |
| **DSR路由协议代码分析** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | 学号 | 姓名 | 班级 | 负责模块 | 成绩 | | 201692454 | 高江 | 软网1603 | 1.代码逻辑分析  2.数据结构：存储单元整理  3.PPT代码讲解  4.核心功能函数汇总 |  | | 201692469 | 武雨桐 | 软网1603 | 1.引言书写  2.PPT制作以及讲解  3.辅助函数汇总  4.DSR基本操作分析 |  | | 201692181 | 陈宇 | 软网1603 | 1.全局变量整理  2.DSR特点分析  3.数据结构：通信单元整理  4.文献搜集，阅读  5.文档排版 |  | |

目录

[第一章 引言 3](#_Toc29292)

[第二章 协议操作介绍概述 4](#_Toc30714)

[2.1 基本DSR路由发现 4](#_Toc9293)

[2.2 基本DSR路由维护 4](#_Toc9934)

[2.3路由发现的附加特点 5](#_Toc3471)

[2.3.1缓存侦听的路由信息 5](#_Toc16497)

[2.3.2 使用缓存路由响应路由请求 5](#_Toc7309)

[2.3.3 路由请求跳数限制 6](#_Toc27491)

[2.4 路由维护的附加特点 7](#_Toc20753)

[2.4.1 数据包回收 7](#_Toc30314)

[2.4.2 在中断链路上的排队数据包 7](#_Toc23648)

[2.4.3 路由自动缩短 7](#_Toc23784)

[2.4.4 增加“路由错误”消息的传播 8](#_Toc26310)

[第三章 数据结构 9](#_Toc32443)

[3.1 通信单元dar\_pkt 9](#_Toc20061)

[3.1.1 in\_addr结构体 10](#_Toc12393)

[3.1.2 dsr\_xxx\_opt结构体 10](#_Toc4382)

[3.1.3 skb 11](#_Toc11622)

[3.2 列表衍生、存储单元 12](#_Toc25735)

[3.2.1 tbl 12](#_Toc1451)

[3.2.2 rreq\_tbl\_entry结构体 12](#_Toc20309)

[3.2.3 send\_buf\_entry结构体 12](#_Toc10501)

[3.2.4 maint\_entry结构体 13](#_Toc32289)

[3.2.5 maint\_buf\_query结构体和link\_query结构体 13](#_Toc1462)

[3.2.6 lc\_node结构体、lc\_link结构体、和cheapest\_node结构体 13](#_Toc4585)

[第四章 全局变量 15](#_Toc7531)

[第五章 函数详解 16](#_Toc24516)

[5.1 核心功能函数 16](#_Toc19089)

[5.1.1 dsr\_rreq\_route\_discovery函数 16](#_Toc4945)

[5.1.2 dsr\_opt\_recv函数 17](#_Toc392)

[5.1.3 dsr\_srt\_opt\_recv函数 18](#_Toc29102)

[5.1.4 dsr\_rreq\_opt\_recv函数 20](#_Toc29773)

[5.1.5 dsr\_rrep\_opt\_recv函数 22](#_Toc10476)

[5.1.6 dsr\_rerr\_opt\_recv函数 23](#_Toc29837)

[5.1.7 dsr\_srt\_shortcut函数 23](#_Toc14429)

[5.1.8 dsr\_rerr\_send函数 25](#_Toc3908)

[5.1.9 dsr\_recv函数 25](#_Toc8248)

[5.1.10 dsr\_dev\_xmit函数 27](#_Toc30289)

[5.2所有函数及其功能汇总 28](#_Toc15197)

[5.2.1 核心功能 28](#_Toc12195)

[5.2.2辅助工具函数、专有操作方法 29](#_Toc31812)

# 引言

动态源路由协议(DSR)是专为移动节点的多跳无线点对点网络而设计的一种简单高效的路由协议。DSR允许网络完全自组织和自配置，而不需要任何现有的网络基础设施或管理。该协议由两个主要的机制组成：“路由发现”与“路由维护”，它们共同工作，允许节点在点对点模式网络中发现和维护到达任意目的地的路由。

该协议的所有方面都是按需操作的，允许DSR的路由数据包开销自动扩展到只需要对当前正在使用的路由的变化作出反应。该协议允许多条路由到任何目的地，并允许每个发送方选择和控制用于路由其数据包的路由，例如用于负载平衡或增加其坚固性。DSR协议的其他优点包括：易于保证无环路路由，在包含单向链路的网络中运行，在路由中只使用“软状态”，当母线在网络中变化时能够非常快速的恢复。DSR协议主要用于高达200节点的移动自组织网络，并且在非常高的移动速率下能够很好地工作。

DSR的基本版本使用显式的“源路由”，其中每个发送的数据包都在其报头中包含一个数据包将通过的完整的、有序的节点列表。这种显式源路由的使用允许发送方选择和控制用于其自身数据包的路由，支持使用多条路由到任何目的地（例如，用于负载平衡），并允许简单地保证所使用的路由是无循环的；通过将此源路由包括在每个数据包的报头中，其他转发或无意听到任何数据包的节点也可以轻松地缓存此路由信息以供将来使用。

# 协议操作介绍概述

本节概述DSR协议的操作。DSR的基本版本使用显式的“源路由”，其中每个发送的数据包都在其报头中包含一个数据包将通过的完整的、有序的节点列表。这种显式源路由的使用允许发送方选择和控制用于其自身数据包的路由，支持使用多条路由到任何目的地（例如，用于负载平衡），并允许简单地保证所使用的路由是无循环的；通过将此源路由包括在每个数据包的报头中，其他转发或无意听到任何数据包的节点也可以轻松地缓存此路由信息以供将来使用。

## 2.1 基本DSR路由发现

当某个源节点发出一个寻址到某个目标节点的新数据包时，源节点在数据包的报头中放置一个“源路由”，给出数据包到达目的地时要遵循的跳数序列。通常情况下，发送者将通过搜索其先前获得到的路由的“路由缓存”来获得合适的源路由；如果它的缓存中没有找到路由，它将启动路由发现协议来动态地找到该目标节点的新路由。在这种情况下，我们将源节点称为“启动器”，目标节点称为路由发现的“目标”。

发起路由发现时，发送节点将原始数据包(触发发现)的副本保存在称为“发送缓冲区”的本地缓冲区中。发送缓冲区包含该节点无法传输的每个数据包的副本，因为它还没有到数据包的目的地的源路由。发送缓冲区中的每个数据包在逻辑上与将其放入发送缓冲区的时间相关联，并且在发送缓冲区中停留一段时间之后，将被丢弃。如果有必要防止发送缓冲区溢出，则还可以使用FIFO或其他替换策略在数据包过期之前将其排除。

当数据包保留在发送缓冲区中时，节点应该偶尔为数据包的目的地地址启动一个新的路由发现。但是，节点必须限制相同地址的这种新路由发现的启动速率，因为有可能目标节点目前无法到达。特别是由于无线传输范围有限以及网络中节点的移动，网络有时会被分割，这意味着目前没有可以通过这些节点转发数据包以到达目的地的节点序列。取决于网络中节点的移动模式和密度的网络分区可能是罕见的，也可能是常见的。

如果一个新的路由发现被在这样一个分区网络中的节点发送的每个数据包都启动了，那么大量的非生产性路由请求数据包将被传播到从此节点可到达的adhoc网络。为了减少这些路线发现的开销，节点应该使用指数后退算法来限制它对同一目标发起新路由发现的比率，并在为同一目标启动的每个连续发现之间加倍超时。如果节点试图超出限制的更频繁地向同一目标节点发送其他数据包，随后的数据包则应在发送缓冲区中缓冲，直到接收到一个到达此目的地的路由的路由应答。但是，在该目标的新路由发现之间的最小允许间隔达到之前，节点不能发起新的路由发现，这种对同一目标的最大路由发现速率的限制类似于在互联网节点上所要求的限制对任何单个目标IP地址发送ARP请求的速率的机制。

## 2.2 基本DSR路由维护

当使用源路由发起或转发数据包时，发送该数据包的每个节点负责确认数据可以通过该链路从该节点流到下一跳。ACK可以提供链路能够承载数据的证明，而在无线网络中，ACK通常是或作为使用中的MAC协议的现有标准部分（例如IEEE802.11定义的链路层确认帧），或通过一个“passive ACK”而被免费提供。如果内置确认机制不可用，则发送数据包的节点可以显式请求在路由沿线的下一个节点返回特定于DSR的software ACK；此software ACK通常将直接发送至发送节点，但是如果这两个节点之间的链接是单向的，这个software ACK则可以在不同的多跳路径上传输。在从某个邻居节点收到ACK后，节点可以选择在一定时间内不需要来自该邻居节点的ACK，除非连接到该邻居节点的节点的网络接口总是接受响应于单播通信的ACK。当一个software ACK已经被应用后，此确认请求应该被重传至最大次数。确认请求的重传可以作为单独的数据包发送，可以在原始数据包的重传上进行备份，或者在不包含software ACK的下一跳目的地的任何数据包上进行备份。在确认请求达到重传的最大次数后，如果还未收到ACK，则发送方将指向下一跳目的地的链接视为当前“中断”。它应该从路由缓存中删除这个链路，并应该向每个从ACK被最后一次接受到以来，已经在该链路上发送数据包的节点返回一个“路由错误”。否则，它应该为这个目标执行一个新的路由发现（但必须符合2.1中的后退规则）。

## 2.3路由发现的附加特点

### 2.3.1缓存侦听的路由信息

节点转发或以其他方式侦听到任何数据包时，应将来自该数据包的所有可用路由信息添加到其自己的路由缓存中。数据包中路由信息的有用性取决于物理介质的方向性特征，以及正在使用的MAC协议。具体而言，有三种不同的情况：

*-网络中的链路通常只能单向(而不是双向)运行，并且在网络中使用的mac协议能够在单向链路上传输单播数据包。*

*-网络中的链接有时只能单向(而不是双向)运行，但是这种对任何链路的单向限制都不是永久的，几乎所有的链路都是物理上的双向的，网络中使用的mac协议能够在单向链路上传输单播数据包。*

*-在网络中使用的mac协议不能在单向链路上传播单播数据包；只能使用双向链路传输单播数据包。例如，ieee 802.11分布式协调功能(DCF)mac协议只能在双向链路上传输单播数据包，因为mac协议要求从接收方返回链路级别的确认数据包，并且还需要在传输节点与接收节点之间进行RTS与CTS数据包的双向交换。*

在上面的第一种情况下，数据包使用的源路由，路由请求中的累计路由记录或路由应答中返回的路由都应该被任何在“前向”方向上的节点缓存。任何节点都应该从接收到的任何这样的数据包中缓存这些信息，无论数据包是寻址到此节点，还是发送到广播(或多播)MAC地址，还是在节点的网络接口处于杂乱模式时被侦听。但是，在这些数据包报头中标识的链接的“反向”方向不应该被缓存。

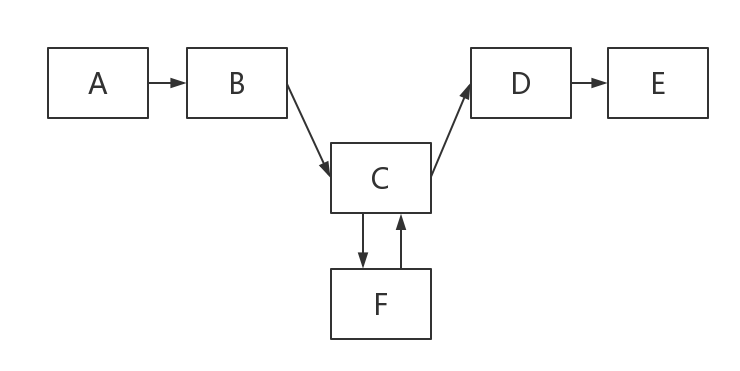
在上面的第二种情况中，链路有时可以单向地运行，应该在两个方向上缓存。

在最后一种情况下，MAC协议要求单播操作具有物理的双向性，则源路由的链路应该在两个方向缓存，除非数据包还包含路由应答，在这种情况下，只应该缓存在此源路由中已经遍历的链路，但不应该缓存该路由中尚未遍历的链路。

### 2.3.2 使用缓存路由响应路由请求

一个接收到不以它为目标的路由请求的节点搜索其自己的路由缓存以寻找到请求的目标的路由。如果找到，该节点通常会将路由回复返回给启动器本身，而不是转发路由请求。在路由应答中，该节点设置路由记录以列出转发路由请求副本的跳数序列，并与从它自己的路由缓存获得的目标的源路由连接。

但是，在发送以这种方式使用其路由缓存中的信息生成的路由应答包之前，节点必须验证在此连接之后路由应答中返回的不包含路由记录中列出的重复节点的结果路由。例如，下图说明节点F已收到针对目标E的路由请求，而节点F在其路由缓存中已经有一条从自身到E的路由：



路由请求

路由 : A - B - C - F 缓存: C - D - E

将来自路由请求的累积路由记录与来自F的路由缓存的缓存路由连接起来，将包括一个从C传递到F并返回到C的重复节点。

在这种情况下，节点F可以尝试编辑路由，以及消除重复，得出从A到B，到C，到D，继续到E的路由，但是在这种情况下，节点F将不会出现在它自己的路由应答中返回的路由上。DSR路由发现禁止节点F从其缓存中返回这样的路由应答；这种禁止增加了产生的路由有效的概率，因为在这种情况下，如果路由之前已经停止工作，节点F将会收到一个“路由错误”。此外，这一禁令意味着，将来遍历路由的“路由错误”很可能通过发送路由回复的任何节点(包括节点F)，这有助于实现确保及时从缓存(如在 F)中删除陈旧数据；否则，由A发起的下一条路由发现，也有可能被F的载有相同的陈旧路由的路由应答所污染。由于基于路由缓存的信息返回路由应答的限制，如果节点F不返回这样的路由应答，节点F通常将会传播路由请求。

### 2.3.3 路由请求跳数限制

每个路由请求消息都包含一个“跳数限制”，可以用来限制允许转发路由请求副本的中间节点的数量。这种跳数限制是使用承载路由请求的数据包中的IP报头中的生存时间字段来实现的。当请求被转发时，这个限制就会减少，如果在找到目标之前限制达到零，那么请求数据包就会被丢弃。该路由请求跳数限制可用于在路由发现尝试期间各种控制路由请求传播的算法的实现。

例如，节点可以使用此跳数限制来实现作为路由发现的初始阶段的“非传播”路由请求。使用这种技术的节点发送其对某个目标节点的第一个路由请求尝试，其跳数限制为1，这样，接收到路由请求的初始传输的任何节点都不会通过重新广播将请求转发到其他节点。这种形式的路由请求称为“非传播”路由请求；它提供了一种廉价的方法，用于确定目标当前是否是发起者的邻居，或者邻居节点是否有到达缓存的目标的路由（有效地使用邻居的路由缓存作为发起者自己的路由缓存的扩展）。如果在短超时后没有收到路由回复，则该节点为目标节点发送“传播”路由请求（即具有由DiscoveryHopLimit配置变量的值定义的跳数限制）。

作为另一个例子，节点可以使用这个跳数限制来实现对目标的“扩展环”搜索。如上文所述，使用此技术的节点发送初始的非传播路由请求；如果没有收到它的路由应答，则节点发出另一个路由请求，其跳数限制为2。对于发起的每个路由请求，如果没有收到路由应答，则该节点会将上次尝试中使用的跳限制加倍，以不允许路由请求在整个网络上传播的条件下逐步探索目标节点。然而，这种扩展环搜索方法可能会增加路由发现的平均延迟，因为在发现到达目标节点的路由之前，可能需要多次发现尝试和超时操作。

## 2.4 路由维护的附加特点

### 2.4.1 数据包回收

当转发包的中间节点通过路由维护检测到该数据包的路由上的下一跳中断时，如果该节点在其路由缓存中有另一条路由到该数据包的目的地，则该节点应该“回收”该数据包，而不是丢弃它。为了挽救数据包，节点将数据包上的原始源路由替换为其路由缓存中的路由。然后，该节点将该数据包转发到沿此源路由指示的下一个节点。在回收一个数据包时，在它被回收的次数中会保持一个计数器，以防止一个包被无休止的回收。否则，由于每个节点只减少以此生存时间数（TTL），因此单个节点可以无限次回收一个数据包。我们选择在每次回收时减少生存时间数，回收数据包也是一项昂贵的操作，所以希望将数据包可以被回收的最大次数与数据包可以被遍历的最大跳数分开绑定。

### 2.4.2 在中断链路上的排队数据包

当转发包的中间节点通过路由维护检测到该数据包的路由上的下一跳链路中断时，除了处理为路由维护而定义的数据包之外，节点还应该以类似的方式处理它在这个新中断的链路上排队的所有挂起的数据包。具体来说，节点应该在其网络接口队列和维护缓冲区中搜索下一跳链路是该新中断链路的数据包。对于当前在此节点排队的每个此类数据包，节点应按以下方式处理该数据包：

*-从节点的网络接口队列和维护缓冲区中删除数据包。*

*-向数据包的原始发件人发出此数据包的“路由错误”，就像该节点已经达到了为路由维护而对该数据包进行重传尝试的最大次数。但是，在为排队的数据包发送这种路由错误以响应检测到的单个新中断链路时，节点应向这些数据包的每个原始发送方发送至多一个“路由错误”。*

*-如果该节点在其路由缓存中有另一条路由到该数据包的IP目的地地址，则该节点应该回收该数据包。否则，节点应丢弃数据包。*

### 2.4.3 路由自动缩短

如果不再需要路由中的一个或多个中间节点，则可以自动缩短正在使用的源路由。这种自动缩短使用中的路由的机制有点类似于“passive ACK”的使用。特别是，如果节点能够侦听到包含源路由的数据包（例如，通过在杂乱接收模式下操作它的网络接口），则此节点将检查该源路由的未扩展部分。如果此节点不是数据包的预定下一跳目的地，而是在数据包源路由的稍后未扩展部分中命名的，则可以推断源路由中自身之前的中间节点不再需要。当决定是否以这种方式返回一个无偿的路由应答时，节点可能会考虑额外的信息，而不仅仅是它能够侦听到数据包。例如，只有当接收到的信号强度或信噪比超过特定阈值时，节点才决定返回无偿路由应答。此外，每个节点维护一个无偿路由应答表，以限制其对同一返回路由的无偿路由回复的比率。

### 2.4.4 增加“路由错误”消息的传播

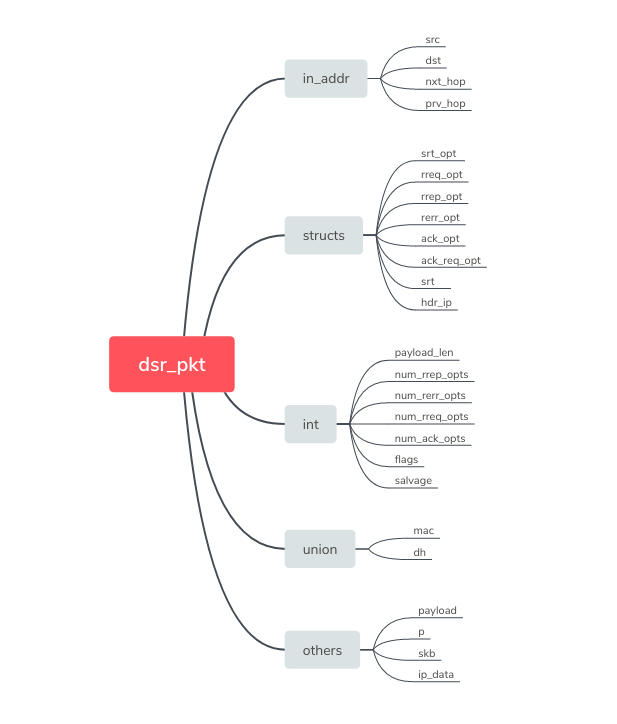
当源节点接收到它发起的数据包的路由错误时，此源节点通过在其下一个路由请求上支持它将该“路由错误”消息传播到其邻居。这样，源节点周围节点缓存中的陈旧信息将不会生成包含与此源节点接收路由错误相同的无效链路的路由应答。

例如，源节点从某节点处的路由错误消息中了解到，从该节点到下一节点的链路当前已中断。因此，它从自己的路由缓存中删除了这个链路，并启动了一个新的路由发现(如果它在路由缓存中没有到目的节点的其他路由)。在发起此路由发现的路由请求数据包上，源节点携带此“路由错误”的副本，确保“路由错误”传播到其他节点，并保证它在响应此路由请求时收到的任何路由答复(包括来自其他节点的路由缓存)都不包含假定存在此断开链路的路由。

# 第三章 数据结构

## 3.1 通信单元dsr\_pkt

不考虑在宏定义了NS2的情况时时，结构体的整体框架如图2.1-1所展示。它包含了各个模块的opt涉及不同的操作。



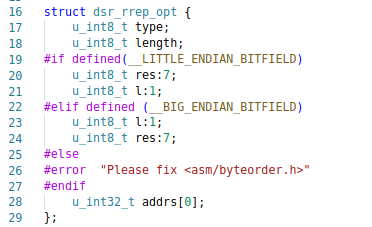
2.1-1 dsr\_pkt结构体

### 3.1.1 in\_addr结构体

in\_addr结构体为系统所定义用于描述一个设备的ipv4地址。in\_addr.s\_addr为unsigned long类型是ip地址实际对应的32位值，是代码中多处进行地址比对的原理。

### 3.1.2 dsr\_xxx\_opt结构体

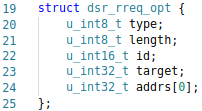
不同的操作对应不同的opt数据结构，对于某一些变量它们大体结构基本类似。以dsr\_rrep\_opt为例举出此种类型的大体存储结构。



dsr-rrep.h

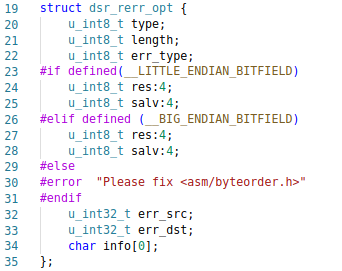
017-024行 使用3个8位无符号整型来携带各种特殊的flag, type等信息

028-028行 使用32位无符号整形进行地址的存储



dsr-rreq.h

022-023行 对于rreq来说，它与rrep的区别是增加了一个以16位无符号数为基本类型的id、和以32为无符号整形为基本类型的target。它们有着不同的功能在之后的函数功能介绍中会有具体的体现。



dsr-rerr.h

032-034行 对于dsr\_rerr\_opt而言，则添加了错误出现的节点位置和错误的类型。

考虑到结构非常类似且篇幅有限，过多的数据结构不在一一列举。省略的数据结构有dsr\_ack\_req\_opt和dsr\_ack\_opt分别对应acknowledgement request 和 acknowledgement 两个网络动作。

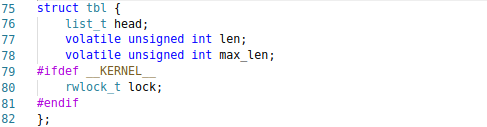
### 3.1.3 skb

skb是sk\_buff的缩写。中文可被译作套接字缓冲区。主要用来在linux网络子系统中各层之间数据传递，起到了“神经中枢”的作用。当发送数据包时，linux内核的网络模块必须建立一个包含要传输的数据包的sk\_buff,然后将sk\_buff传递给下一层，各层在 sk\_buff 中添加不同的协议头，直到交给网络设备发送。同样，当接收数据包时，网络设备从物理媒介层接收到数据后，他必须将接收到的数据转换为sk\_buff，并传递给上层，各层剥去相应的协议头后直到交给用户。

## 3.2 列表衍生、存储单元

### 3.2.1 tbl

tbl为table的缩写，有点与不足在下面分析，此种类型定义的变量有rreq\_tbl，grat\_rrep\_tbl，send\_buf，neigh\_tbl，maint\_buf由各个节点来维护。



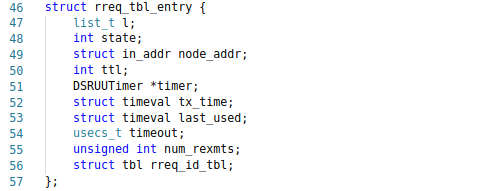
tbl.h

076-076行 tbl结构体由list\_t封装而来

080-080行 定义程序锁避免不同进程之间读取时出错

077-078行 定义当前长度和最大长度在list\_t的基础上增加list\_t的安全性

### 3.2.2 rreq\_tbl\_entry结构体



dsr-rreq.c

047-056行 在该结构体中定义了状态，时间等一系列参数。这个结构体的功能包括判断超时退出。定义了ttl表示当前的跳数，定义了节点的地址等。每一个对象用来描述一次路由发现的运行。主要用于计时，统计TTL等。所有该类型的变量自创建起都保存于rreq\_tbl之中。其中元素具有唯一性。

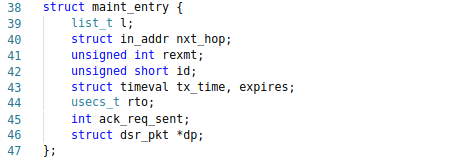
### 3.2.3 send\_buf\_entry结构体

send-buf为发送缓冲，每个包发送之前都要先压入缓冲由设备来进行发送。用send-buf-entry来描述一次buf sending可以避免由于设备故障等因素造成的过长时间的等待和资源浪费。

 send-buf.c

### 3.2.4 maint\_entry结构体

maint-entry和maint-buf-query为maint-buf操作过程中所运用的两个结构体。在一个动态的网络中，节点状态往往会随时间的变化而发生改变，所以maint-buf需要定期的维护。使用maint-entry定期检测与邻居节点的连通性可以保证当与邻居链路断开时，路由错误信息立即被发布。保障了连接的稳定性。每个maint-entry用来描述一次路由维护的过程。

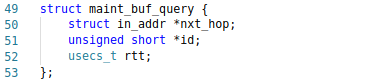


maint-buf.c

所有的该类型变量被收集与maint-buf之中。函数清单中列出了对maint-buf的各类基本操作对应的函数。

### 3.2.5 maint\_buf\_query结构体和link\_query结构体

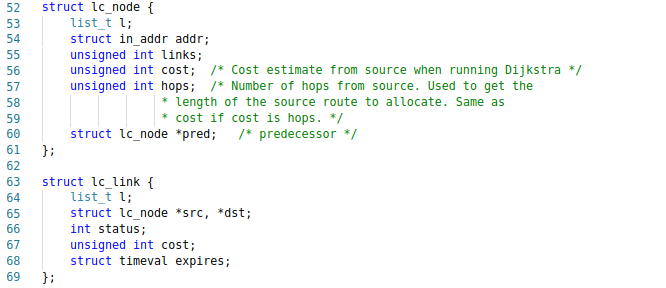
我们使用一个maint\_buf\_query来描述对一个目的节点的询问。当一条链路断开时，我们可以利用该结构体查询所有与节点相关的路径信息，并立即清空。

maint-buf.c

类似的结构link\_query和以上结构体功能类似。

### 3.2.6 lc\_node结构体、lc\_link结构体、和cheapest\_node结构体

与maint\_buf不同的是，lc(linc-cache)中存有两个list\_t列表。分别对应节点和边集。根据地杰斯特拉算法我们能够通过lc来得到所有联通节点的最短路径信息。lc\_node, lc\_link的功能便是如此。而cheapest\_node用来存放花销最小的节点，用来辅助迪杰斯特拉等算法的运行。





link-cache.c

# 全局变量

全局变量由列表形式给出，并概括了其具体功能。

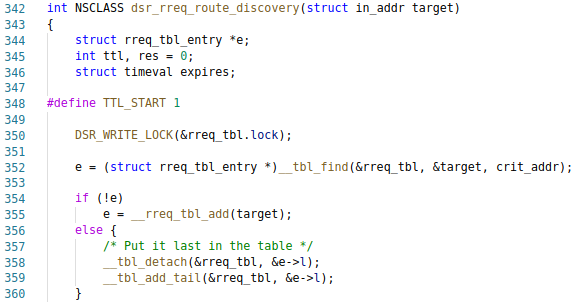
|  |  |  |
| --- | --- | --- |
| **变量名称** | **功能** | **声明位置** |
| dsr\_node | dsr节点 | dsr.h |
| rreq\_tbl | 整理存放路由发现描述符 | ns-agent.c |
| grat\_rrep\_tbl | 整理存放路由回复 |
| send\_buf | 整理存放发送入口描述符 |
| neigh\_tbl | 邻居表 |
| maint\_buf | 整理存放路由维护描述符 |
| grat\_rrep\_tbl\_timer | 计时器 |
| send\_buf\_timer |
| neigh\_tbl\_timer |
| lc\_timer |
| LC | link-cache | link-cache.c |

# 函数详解

在函数详解一章的讲解中，我们挑出了一系列比较核心的函数进行了具体的讲解。对于函数讲解而言，我们在保证整体结构完整的前提下略去一些太过细节的部分。目的是更加清晰的刻画整个协议的大体流程。

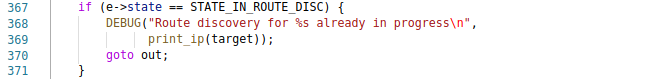
## 5.1 核心功能函数

### 5.1.1 dsr\_rreq\_route\_discovery函数



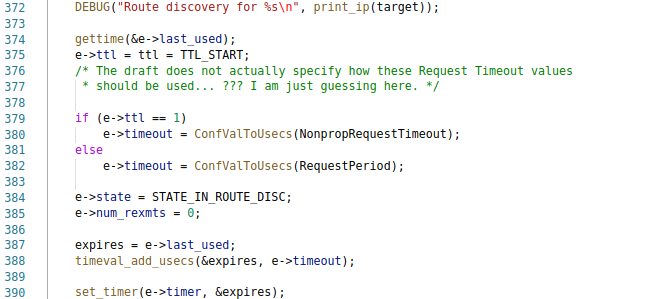
344-344行 e是一个rreq\_tbl\_entry类型的指针。

352-352行 使用\_\_tbl\_find函数，对rreq\_tbl进行搜索。如果列表中存在有该过程，则将之移动至末尾。



dsr-rreq.c

367-371行 如果同时出现两个发现同一节点的路由发现(discovery)第一个影响效率第二个有可能会引发错误。在第367行进行判断满足条件则跳转至末尾。



375-390行 在准备过程中，函数初始化时间，ttl等值。

dsr-rreq.c

394-394行 准备完成后，通过掉用相应函数，在网络启动discovery

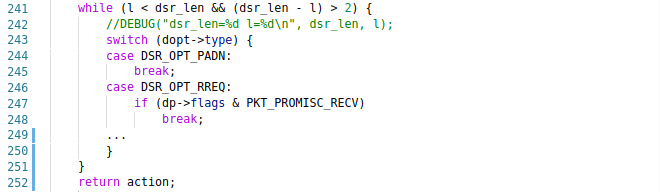
### 5.1.2 dsr\_opt\_recv函数



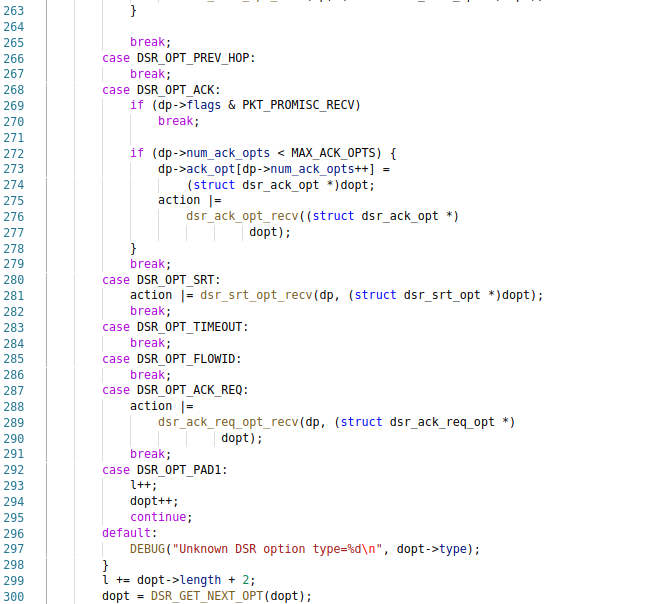
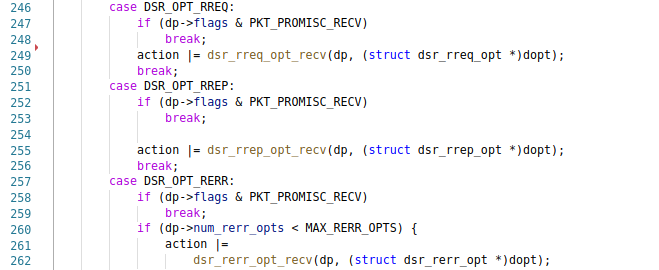
略去了初始化及其细节部分，218行是dsr\_opt\_recv进行的第一个操作。使用if条件判断进行安全性验证。



action是一个标记，可以用来存放当前进行操作的类型。如果该数据包目的地是本节点并且负载长度不为0，则将action设置为DSR\_PKT\_DELIVER宏值。



dsr-opt.c251-252行不同数据包对应于不同种类的接收方式,对应返回不同的action

 dsr-opt.c

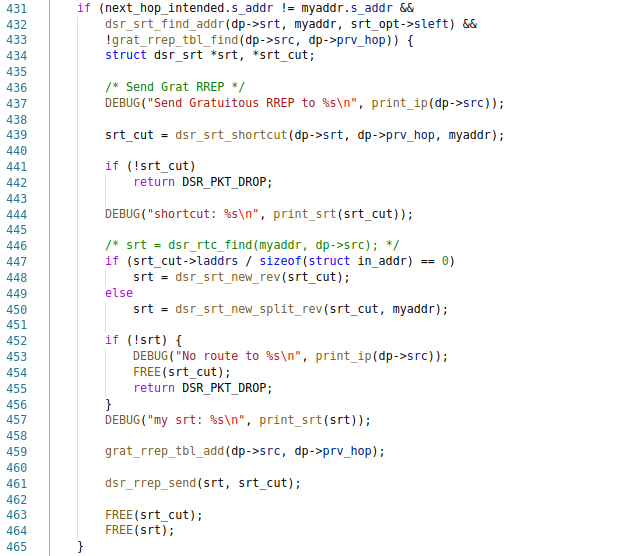
在父级函数中执行了DSR\_GET\_OPT宏并取出dp数据包中的数据包类型。在此用switch 来对不同数据包进行不同的接收操作。执行结果既返回了action，并且将数据包接收至dp指针处。不同数据包对应于不同种类的接收方式,对应返回不同的action值。

其中dsr\_srt\_opt\_recv能够根据接收的内容，回复对应的reply(dsr\_rrep)而不用再回到dsr\_opt\_recv函数之中执行。类似的dsr\_ack\_req\_opt\_recv会自动回复dsr\_ack。

### 5.1.3 dsr\_srt\_opt\_recv函数

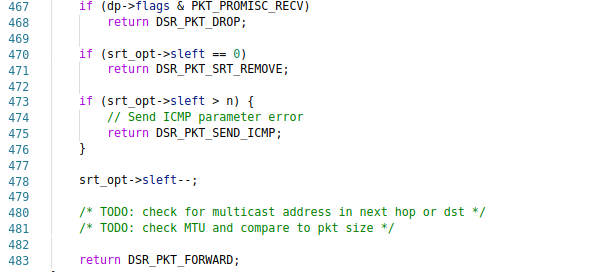


当dsr\_opt\_recv函数判断当前操作为DSR\_OPT\_SRT时，该函数既被调用。本函数调用的dsr\_srt\_new将源路由信息加入到cache之中。



dsr-srt.c

431-464行进行了一系列操作包括地址比对，检索列表等。寻找到需要进行路由缩短的位置。如果包中的源路由路径可以用来优化目前。则调用dsr\_srt\_shortcut函数，从dp->srt中得到一条从dp->srt到dp->prv\_hop的路由。使用grat\_rrep\_tbl\_add，进行路由reply消息的发送，通知前列节点路径信息已经被缩短。本地不会缓存缩短后的路径信息，路由信息由前置节点存储。



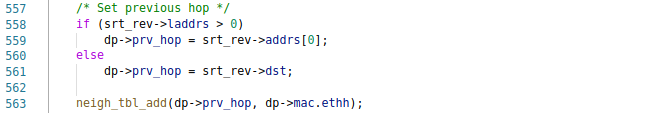
dsr-srt.c

467-483行 完成操作后，设定一系列规则返回不同值。代表上层函数会执行不同的操作。

### 5.1.4 dsr\_rreq\_opt\_recv函数



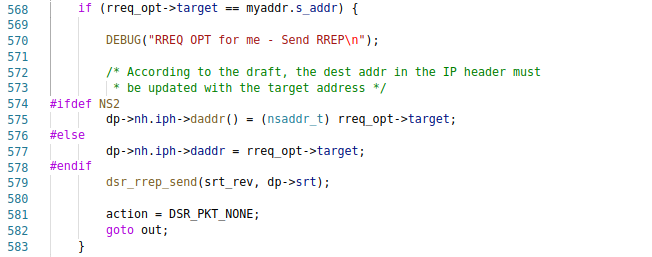
在接收过后将数据包传入该函数之中。避开所有的判断安全性和初始化的部分，对代码进行解析。首先添加该节点至rreq\_tbl之中。



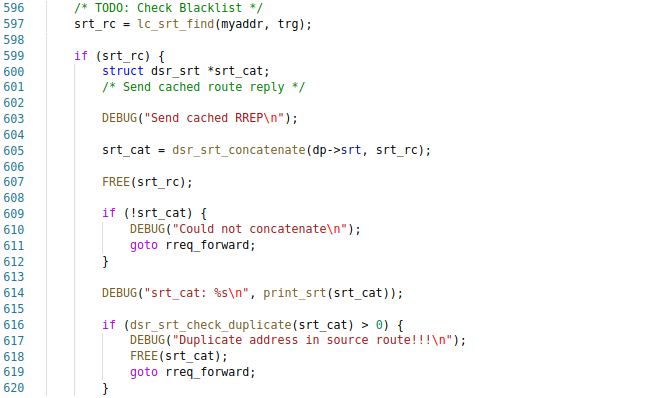
dsr-rreq.c

563-563行 rreq\_tbl\_add\_id会将新添加的节点放在列表最末尾

559-561行srt\_rev中，存有本身生成的反向路由。

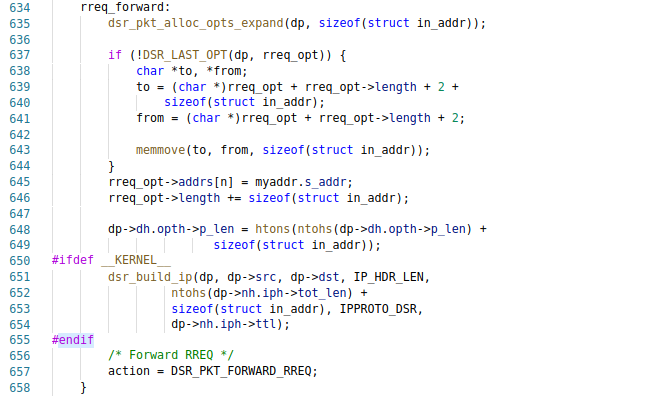


579-582行 通过反向的路径信息回传rrep消息。当消息回复以后，立即跳转，返回上层函数。



dsr-rreq.c

如果函数未跳转，程序会继续执行。dsr\_srt\_concatenate函数可以将路由串联起来，将本地路由连接到原路由之后。



dsr-rreq.c

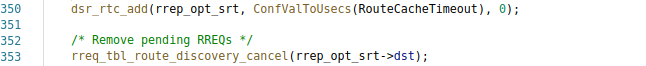
在条件满足时调用dsr\_srt\_concatenate函数处理路径信息。若条件不满足，则跳转到标签处开始执行。经过该部分action被赋值为DSR\_PKT\_FORWARD\_RREQ并将于函数最末尾返回。

### 5.1.5 dsr\_rrep\_opt\_recv函数

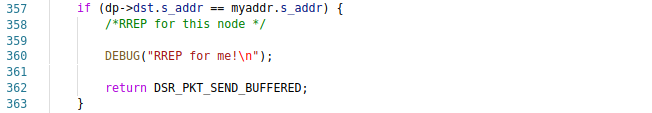
rrep的职责包括对RREP消息的转发与处理。

来看dsr\_rrep\_opt\_recv函数

首先进行路由发现关闭过程并且将搜索到的路径信息缓存。表示一次发现过程已经完成。



然后进行地址比对。

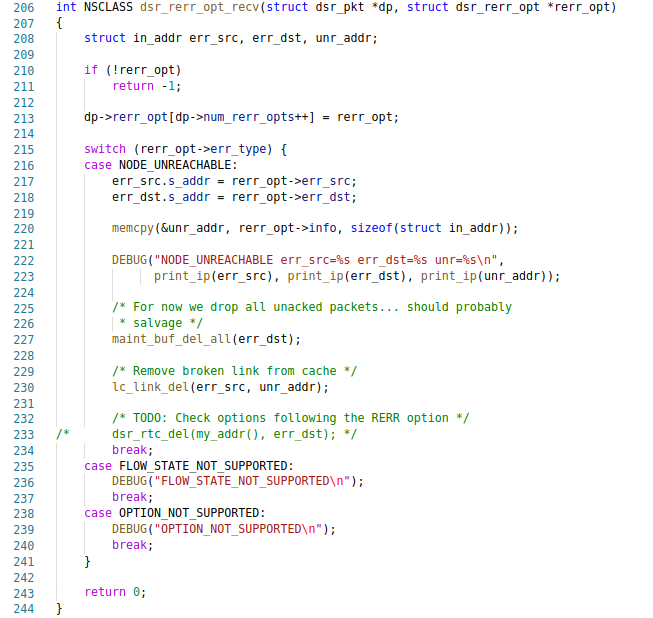


dsr-rrep.c

如果目的地址是本机地址，action返回DSR\_PKT\_SEND\_BUFFERED并且结束程序。如果不是则返回DSR\_PKT\_FORWARD交由上一层处理。

### 5.1.6 dsr\_rerr\_opt\_recv函数

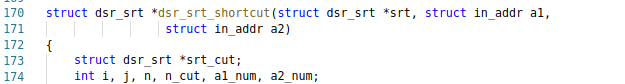
该函数的主要作用是传播路由错误信息。通过调用maint\_buf\_del\_all函数并且触发maint\_buf\_set\_timeout函数，再由maint\_buf\_set\_timeout函数调用dsr\_rerr\_send，对错误进行传输。所有节点收到数据包后，立刻将路由信息从自己的maint-buf中删除，并且帮助进行扩散。



dsr-rerr.c

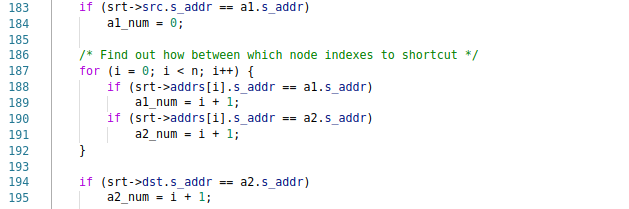
### 5.1.7 dsr\_srt\_shortcut函数

函数传入了一个源路由信息和两个地址信息。并定义了一系列临时变量。



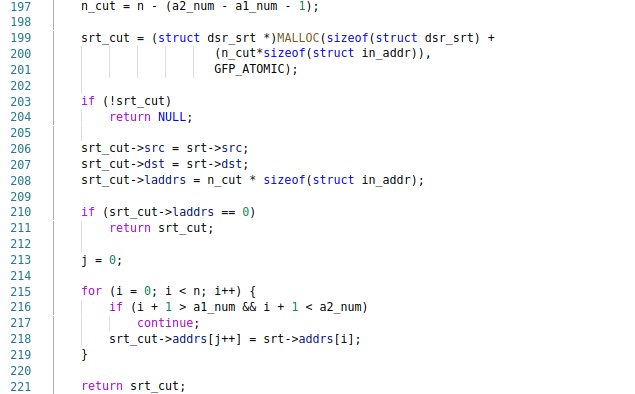
dsr-srt.c

然后寻找a1,a2节点在地址中对应的位置。



dsr-srt.c

最后生成两点之间的源路由并返回。



dsr-srt.c

### 5.1.8 dsr\_rerr\_send函数

设定目的地和事故路段后，调用传输函数将数据包传送至目的地。根据原理进行操作所以略去一部分代码选择功能最核心的部分后得到以下代码段。







dsr-rerr.c

081-084行 将源路由路径填入节点中

186-187行 将源节点地址和目的节点地址填入数据包后调用函数进行包传输

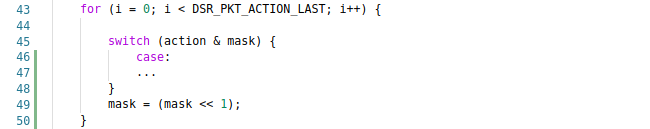
篇幅有限类似函数及其具体功能，不再通过细节展开。将采用列表的形式在之后的一章中呈现。

194-194行 调用定义好的函数进行包传输

### 5.1.9 dsr\_recv函数

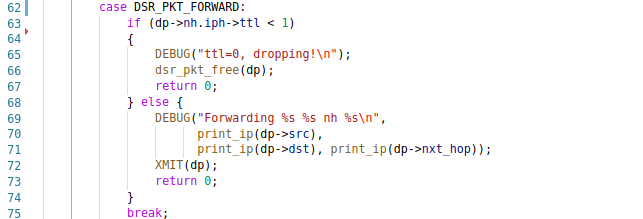


在调用dsr\_opt\_recv之后，dsr\_recv得到了一个完整的数据包（dp）。通过对action的区分，可以对不同的数据包进行不同的处理。dsr\_recv间接调用了以上所有的接收函数，是通过调用dsr\_opt\_recv函数来实现



dsr-io.c

如果收到数据包，并且通过action隐含的信息判断是否需要将该数据包转发。



dsr\_recv函数收到DSR\_PKT\_FORWARD\_RREQ消息之后，会直接将数据包传送并且立即结束运行。

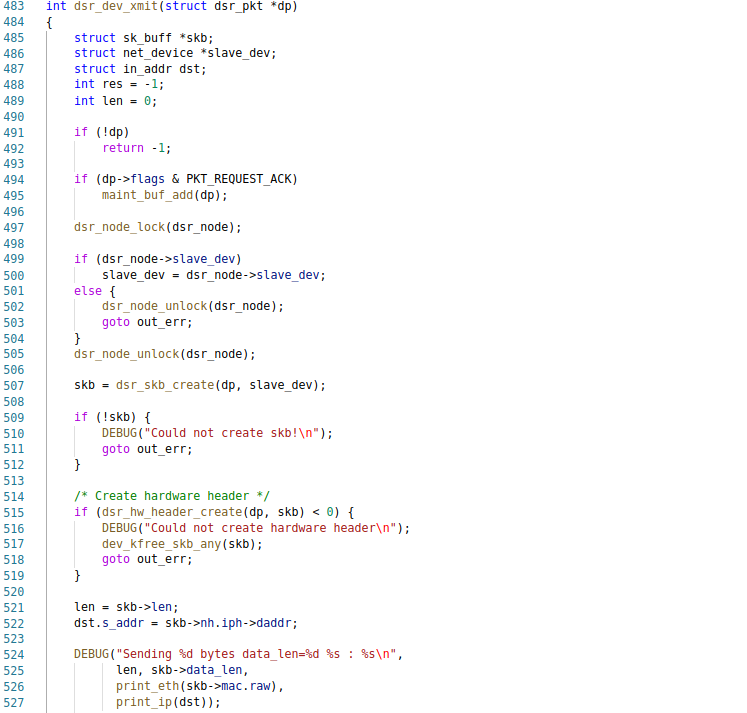


dsr-io.c

077-078行 开始进行传输

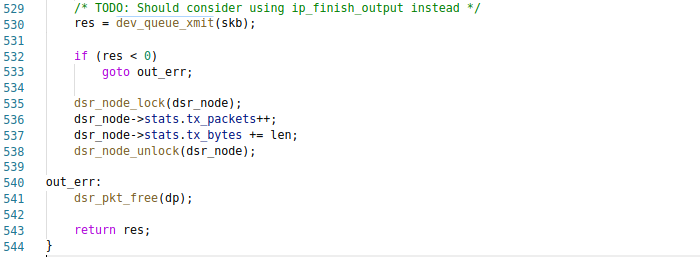
### 5.1.10 dsr\_dev\_xmit函数

所有数据包均由该函数发送。



dsr-dev.c

在完成对skb的各种初始化操作后，调用系统函数dev\_queue\_xmit对skbuff中的内容进行传输。并且释放dp的内存空间。



dsr-dev.c

## 5.2所有函数及其功能汇总

这部分内容使用列表将函数名，函数对应的文件和功能总结列于表格中。主要包括一些具体的数据操作，如向列表中添加、删除元素等。其中的各种函数并不独立，可以存在一定的相互调用关系。我们的主要的工作内容是总结每个函数的具体功能。

### 5.2.1 核心功能

|  |  |  |
| --- | --- | --- |
| 函数名 | 功能（包括但不限于） | 声明位置 |
| dsr\_recv | 接收处理函数 | dsr-io.h |
| dsr\_opt\_recv | dsr-opt.h |
| dsr\_srt\_opt\_recv | dsr-srt.h |
| dsr\_rerr\_opt\_recv | dsr-rerr.h |
| dsr\_rreq\_opt\_recv | dsr-rreq.h |
| dsr\_rrep\_opt\_recv | dsr-rrep.h |
| dsr\_srt\_shortcut | 从srt中得到短路径信息 | dsr-srt.h |
| dsr\_rerr\_send | 发送路由错误 | dsr-rerr.h |
| dsr\_rrep\_send | 发送路由reply | dsr-rrep.h |
| dsr\_rreq\_send | 再网络中启动路由发现 | dsr-rreq.h |
| dsr\_ack\_req\_send | ack请求消息 | dsr-ack.h |
| dsr\_ack\_send | 发送ack | dsr-ack.h |

### 5.2.2辅助工具函数、专有操作方法

|  |  |  |
| --- | --- | --- |
| 函数名 | 功能简介（包括但不限于） | 声明位置 |
| dsr\_pkt\_alloc\_opts | 动态内存分配pkt\_opt | dsr-pkt.h |
| dsr\_pkt\_alloc\_opts\_expand | 扩展pkt\_opt |
| dsr\_pkt\_free\_opts | 释放pkt\_opt |
| dsr\_pkt\_alloc | 动态内存分配pkt |
| dsr\_pkt\_free | 释放pkt |
| dsr\_opt\_hdr\_add | 将缓冲区转换为dsr\_opt\_hdr | dsr-opt.h |
| dsr\_build\_ip | 构建iphdr |
| dsr\_opt\_find\_opt | 传入type，通过type寻找对应的opt |
| dsr\_opt\_remove | 删除pkt的opt |
| dsr\_opt\_parse | 通过一定规则解析opt |
| dsr\_rerr\_opt\_add | 创建一个dsr\_rerr\_opt用来发送路由错误 | dsr-rerr.h |
| crit\_query | 比对内存  作为规则辅助tbl进行表内搜索 | dsr-rrep.h |
| crit\_time |
| crit\_addr | send-buf.h |
| crit\_garbage |
| crit\_addr\_id\_del | maint-buf.h |
| crit\_addr\_del |
| crit\_addr |
| crit\_ack\_req\_sent |
| crit\_expires |
| crit\_addr | dsr-rreq.h |
| crit\_duplicate |
| grat\_rrep\_tbl\_init | rrep\_tbl基本操作 | dsr-rrep.h |
| grat\_rrep\_tbl\_cleanup |
| grat\_rrep\_tbl\_add |
| grat\_rrep\_tbl\_find |
| grat\_rrep\_tbl\_print |
| grat\_rrep\_tbl\_proc\_info |
| grat\_rrep\_tbl\_timeout |
| dsr\_rrep\_add\_srt | 添加source route | dsr-rrep.h |
| \_\_rreq\_tbl\_entry\_create | 创建rreq\_tbl基本单元 | dsr-rreq.h |
| \_\_rreq\_tbl\_add | 添加元素至rreq\_tbl\_add |
| rreq\_tbl\_timeout | 检测超时并进行相应操作 |
| rreq\_tbl\_add\_id | 使用id信息构造rreq\_tbl基本单元并添加 |
| dsr\_rreq\_route\_discovery | 开启一次路由发现 |
| rreq\_tbl\_route\_discovery\_cancel | 关闭特定的路由发现 |
| rreq\_tbl\_set\_max\_len | 得到最大长度 |
| dsr\_rreq\_duplicate | 检测重复，有则返回1，没有返回0 |
| dsr\_srt\_next\_hop | 寻找下一跳地址 | dsr-srt.h |
| dsr\_srt\_prev\_hop | 寻找上一跳地址 |
| dsr\_srt\_find\_addr | 是否包含对应地址有返回1没有返回0 |
| dsr\_srt\_new | 使用一定信息创建一个dsr\_srt |
| dsr\_srt\_new\_rev | 复制一个dsr\_srt |
| dsr\_srt\_new\_split | 返回指定地址之前的路径地址 |
| dsr\_srt\_new\_split\_rev | 从接收的dsr\_srt中分出地址 |
| dsr\_srt\_check\_duplicate | 查重 |
| tbl\_empty | tbl结构体基本操作 | tbl.h |
| \_\_tbl\_add |
| \_\_tbl\_add\_tail |
| tbl\_add\_tail |
| \_\_tbl\_find |
| \_\_tbl\_detach |
| \_\_tbl\_del |
| \_\_tbl\_find\_do |
| tbl\_find\_do |
| \_\_tbl\_do\_for\_each |
| tbl\_do\_for\_each |
| tbl\_find\_detach |
| tbl\_detach |
| tbl\_detach\_first |
| tbl\_add |
| tbl\_del |
| tbl\_find\_del |
| tbl\_for\_each\_del |
| send\_buf\_print | send-buf基本操作 | send-buf.h |
| send\_buf\_set\_max\_len |
| send\_buf\_timeout |
| send\_buf\_entry\_create |
| send\_buf\_enqueue\_packet |
| send\_buf\_set\_verdict |
| send\_buf\_flush |
| send\_buf\_print |
| send\_buf\_get\_info |
| send\_buf\_cleanup |
| maint\_buf\_init | maint-buf基本操作 | maint-buf.h |
| maint\_buf\_cleanup |
| maint\_buf\_set\_max\_len |
| maint\_buf\_add |
| maint\_buf\_del\_all |
| maint\_buf\_del\_all\_id |
| maint\_buf\_del\_addr |
| maint\_buf\_set\_timeout |
| maint\_buf\_timeout |
| maint\_buf\_salvage |
| lc\_link\_del | link-cache基本操作 | link-cache.h |
| lc\_link\_add |
| lc\_garbage\_collect\_set |
| lc\_garbage\_collect |
| lc\_srt\_find |
| lc\_srt\_add |
| lc\_flush |
| lc\_init |
| lc\_cleanup |
| \_\_dijkstra | 迪杰斯特拉运算 |