**《基于层次状态机的水声网络协议栈实现软件》**

**使用设计说明**

# 一、引言

## 编写目的

本说明书为方便研究人员在Linux操作系统平台下进行网络协议仿真或使用实际通信节点进行实地网络协议测试而编写。本程序包含了一般网络协议栈所包含的传输层、路由层、MAC层和物理层。每个层都至少实现了一个协议。开发人员也可以根据自己的需要增加网络协议或修改已有的网络协议。

# 二、软件总体设计

## 软件需求概括

本软件开发方法是：传统的生命周期方法，自顶向下，逐步求精的结构化的软件设计方法。在总体设计框架完成的前提下，可以逐步扩展。

本软件主要包括以下几个主要的功能：

1. 支持驱动多种设备
2. 数据发送以及接收
3. 输出Trace文件以分析网络性能
4. 实时输出协议栈运行日志
5. 使用配置文件
6. 配置协议栈中使用的协议和协议参数

## 软硬件系统配置

1. 硬件设备

该软件适合运行于Ubuntu 16.04.3 LTS及以上任何版本的操作系统上，没有配置高低的要求。

1. 软件环境

该软件的实现是在Ubuntu 16.04.3 LTS系统下安装的 libyaml-cpp-dev、 libboost-all-dev和evlib下实现的。

## 条件与限制

该软件运行于Ubuntu 16.04.3 LTS操作系统下，由于仿真场景的限制，本协议栈最多在2X2X2km的仿真场景中同时运行20个节点。

## 协议栈

### 整体架构

本程序所实现的协议栈也是基于四层协议体系（传输层、路由层、MAC层、物理层）而设计,与传统协议栈不同,本协议栈中的每一层可以存在多种协议,在运行时再通过配置文件选择具体运行的协议。而且为协议之间的跨层通信提供了简单易用的接口。图 2-1是协议栈的总体架构设计图,主要由五个部件组成

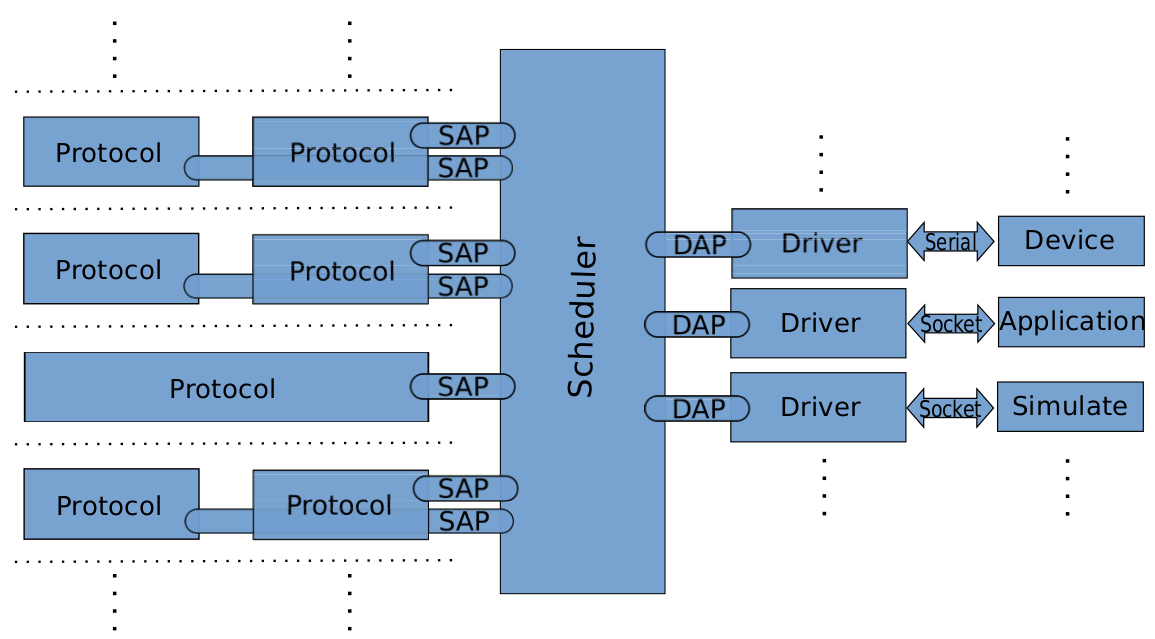


图2-1 协议栈总体架构图（拟更换）

1. 协议 (Protocol)：协议模块,即网络协议栈中各个层次的具体协议的实现,由用户编写，具体的协议实现方法见下节。
2. 调度器 (Scheduler)：协议栈整体架构的消息中转站，各个模块的消息（数据包传递、通知消息、I/O事件等等）都要通过调度器转发给其他模块处理
3. SAP(Service Access Point)：与协议和调度器相连接，用于实现网络协议栈中相邻层的数据包交换及消息通知和协议间跨层通信。
4. 驱动 (Driver)：用于驱动外部设备，使得协议栈能够和真实的水声通信节点交换数据包和通信指令，将数据包发送到真实信道中，进行实地测试;同时也支持驱动仿真信道，通过电脑仿真验证协议可行性。
5. DAP(Device Access Ponit)：与SAP类似，不过DAP连接的不是协议与调度器，而是驱动与调度器

### 模块间通信

**协议栈中的消息类型**

本协议栈中的消息类型分为以下四种：

1. 请求(Request, REQ)
2. 响应(Response, RSP)
3. 通知(Notification, NTF)
4. I/O事件

本协议栈的跨层通信架构是无管理的，任意两个协议间都能够通信，通信采用一种层间通信的规范，把层间通信分为三类：请求 (Request, REQ)、响应(Response, RSP) 和通知 (Notification, NTF)，把通信的双方分别称为客户端 (Client) 和服务端 (Server)。发出请求的是客户端,另外一方是服务端，当服务端收到请求后，必须向客户端返回一个响应;发出通知的是服务端，另外一方是客户端，客户端收到通知后不用回复响应。一般来说，在层间通信中，上层为服务器，下层为客户端。

I/O事件包括以下：

1. 物理层收到数据包：实际的设备收到数据包后交给协议栈处理
2. 传输层收到数据包：协议栈上层具体应用（文本消息、图片等）产生数据包后交给协议栈处理
3. 定时器事件：定时器事件由协议产生

**SAP / DAP 和调度器的作用**

协议栈中每个协议都继承Module模板类,模板类中已经提供了三个接口函数

1. SendReq()
2. SendRsp()
3. SendNtf()

三个接口函数分别对应与层间通信的三类消息分发，协议通过这些接口调用 SAP，从而与其它协议进行层间通信，SAP 连接协议和调度器。当协议使用层间通信的接口时，SAP 把协议的层间通信消息存分门别类放到四个消息队列中。然后调度器从四个消息队列中取出层间通信消息，发送给对应协议的 SAP。调度器工作的整个过程如图2-2所示。

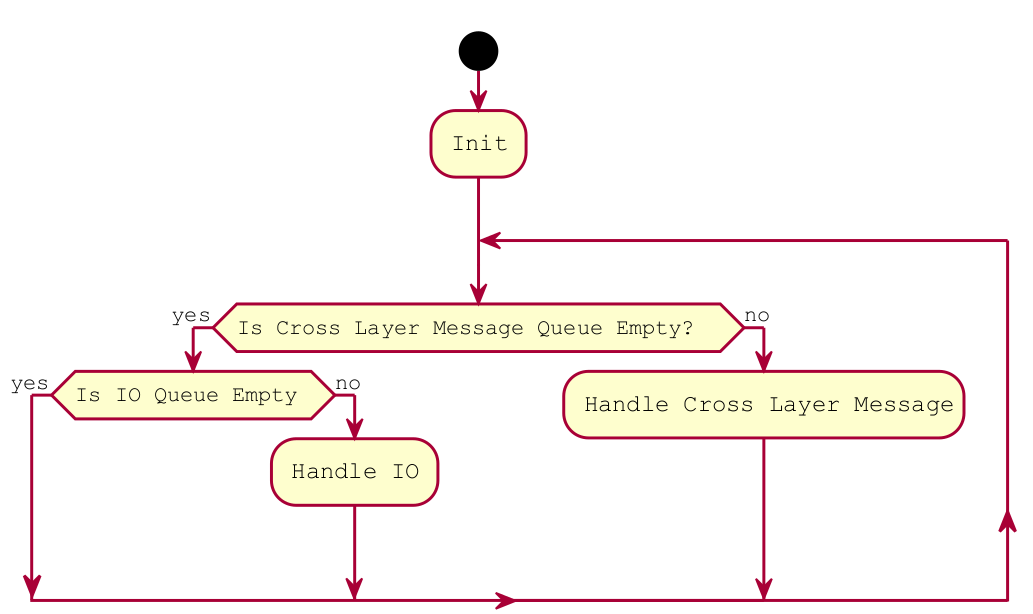


图 2-2 调度器工作过程

本协议栈里使用二元组 (layer, pid) 区分不同的协议以及对应的 SAP，layer 指协议的层次,pid 指该层中协议的 ID，协议栈里使用一个映射表保存二元组和 SAP 实例的映射。由于整个协议栈内只有也只能有唯一一个映射表，所以这里使用了单例模式 (Singleton)来编写这个映射表。单例模式是一种常用的软件设计模式，在整个系统中，单例模式能保证使用该种模式的类的对象有且只有一个。

整个协议栈由事件驱动，在协议栈运行过程中，每当有层间通信或I/O事件事件发生时，调度器调度器就会调用相应的模块执行操作。整个设计和实现都比较简单明了,所需的系统资源也不多,甚至能够运行在一些嵌入式环境中。

### 事件驱动库

本协议栈需要侦听应用层以及物理信道来的数据流，因此需要一个事件驱动库或者多线程多进程等编程模型接收外界的数据。

相比其他模型，使用 select() 的事件驱动模型只用单线程（进程）执行，占用资源少，不消耗太多CPU，同时能够为多客户端提供服务。如果试图建立一个简单的事件驱动的服务器程序，这个模型有一定的参考价值。但这个模型依旧有着很多问题。首先，select()接口并不是实现“事件驱动”的最好选择。因为当需要探测的句柄值较大时，select() 接口本身需要消耗大量时间去轮询各个句柄。很多操作系统提供了更为高效的接口，如 linux 提供了 epoll，BSD 提供了 kqueue，Solaris 提供了 /dev/poll …。如果需要实现更高效的服务器程序，类似 epoll 这样的接口更被推荐。遗憾的是不同的操作系统特供的 epoll 接口有很大差异，所以使用类似于 epoll 的接口实现具有较好跨平台能力的服务器会比较困难。其次，该模型将事件探测和事件响应夹杂在一起，一旦事件响应的执行体庞大，则对整个模型是灾难性的，在很大程度上降低了事件探测的及时性。

幸运的是，有很多高效的事件驱动库可以屏蔽上述的困难，常见的事件驱动库有 libevent 库，还有作为 libevent 替代者的 libev 库。这些库会根据操作系统的特点选择最合适的事件探测接口，并且加入了信号 (signal) 等技术以支持异步响应，这使得这些库成为构建事件驱动模型的不二选择。下面将介绍如何使用 libev 库替换 select 或 epoll 接口，实现高效稳定的服务器模型。

Libev是一个基于Reactor模式的事件库，效率较高、代码精简，是一个值得学习与使用的轻量级事件驱动库。在网络协议仿真平台上，我们主要运行Libev的IO事件侦听，定时器事件以及其prepare机制，该机制是在每次事件循环之前都执行一段代码，在本平台上是执行调度器代码。

本协议栈需要按需处理IO事件以及定时器事件。

其中，IO事件的处理如下：

ev\_io m\_read\_listen;

ev\_io\_init(&m\_read\_listen, read\_listen, listenfd, EV\_READ);

ev\_io\_start(LibevTool::Instance().GetLoop(), &m\_read\_listen);

第一句是声明IO事件的句柄，第二句是设置IO事件的回调函数，关心的文件描述符以及事件，第三句是开启该IO事件的侦听器。

定时器事件通常都是提供给协议使用，因此已经封装好提供给层次状态机使用，通过函数SetTimer(时间值, 定时器ID) 设置定时器时间并开启定时器，定时时间一到，层次状态机会发起一个msg::MsgTimeOut事件，状态机捕捉该事件即可。

### 配置文件

**配置每一层使用的协议**

在本协议栈中，同一网络层次中可能实现了多个不同的协议，例如路由层中就实现了DRT、SRT、DAB等协议，但是在协议栈运行时在每一层中只能同时运行一个协议，因此在协议栈运行之前，需要使用配置文件对每一层的协议进行指定。

本程序使用头文件的宏定义对每一层的协议进行配置，具体配置在moduleconfig.h文件中，该文件如图2-3所示

其中//为注释，首先，分别将传输层、路由层、MAC层和物理层的层数定义为4～1，如”#define MAC\_LAYER (2)“即为将MAC层定义为第二层。其次分别对每一层使用的协议编号进行配置，如”#define MAC\_PID (6)“即为配置MAC层使用的协议ID为6，对应的协议是AlohaSpecial。

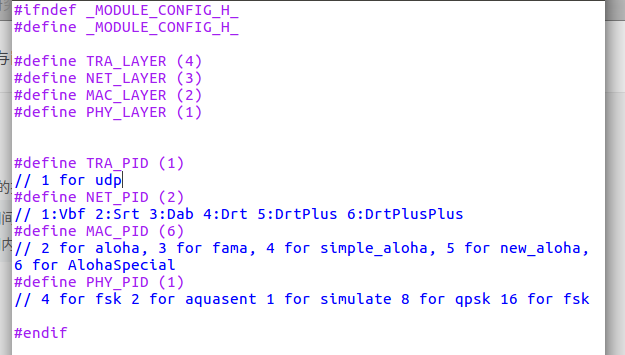


图2-3 moduleconfig.h

**配置协议使用的参数**

协议在测试过程中经常会出现需要修改协议参数的情况，例如MAC层中最简单的Aloha协议，需要配置的参数就有本节点的MAC地址、超时重发时间和重发等待时间等。在没有使用配置文件的情况下，每次修改协议参数都要去修改协议实现的源代码，然后重新编译，运行测试，十分麻烦。为了减少测试过程中修改协议参数过程的工作量，本程序引入了使用配置文件的解决方案。

配置文件使用了YAML语言格式进行编写，YAML 语言的设计目标就是方便人类读写。它实质上是一种通用的数据串行化格式。本程序使用到的配置文件格式如下：

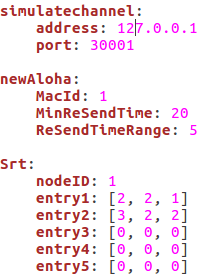


图2-4 配置文件示例

每个节点需要用到的所有配置参数都写在一个文件中，配置文件命名如”configA.yaml“，以”config“后面的字母区别不同的节点使用的配置文件。一个配置文件中包含了多个协议的配置参数，每个协议的配置参数都以固定的形式写在文件中，如图2-4所示，配置文件大小写敏感，使用缩进表示层级关系，缩进时不允许使用Tab键，只允许使用空格，缩进的空格数目不重要，只要相同层级的元素左侧对齐即可。

图2-3中显示了两个协议的配置信息，分别是MAC层newAloha协议和路由层Srt（静态路由）协议，具体的参数信息如下，MacId 表示本节点的MAC地址MinReSendTime 表示最小的重发等待时间，ReSendTimeRange表示重发时间的选择范围。则可以看出，本次newAloha协议配置的超时重发时间为20～25,单位是秒，本节点的MAC地址是1。另外一个协议的配置参数即路由层的Srt协议，与newAloha类似，其配置的节点ID是1，entry1~entry5分别为静态路由表条目。

在协议运行中，需要对配置文件进行参数读取，以mac/new\_aloha.cc源码文件中读取参数为例如图2-5所示。等号左边的selfMacId、minReSendTime、reSendTimeRange分别为newAloha协议运行时从配置文件中读取的参数。

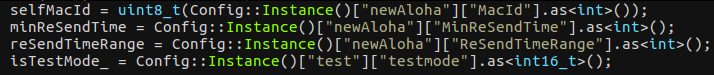


图2-5 读取配置文件参数示例

### trace系统

Trace系统主要是为了追踪协议栈中的数据包，在每个数据包向上传递或者向下传递的过程中，输入关于数据包的日志到文件中。一方面，文件可以用来进行问题分析和定位，另一方面，文件可以用来做性能分析。

Trace模块主要对应于Trace类，其中提供了两个静态重载方法，在代码文件中与数据包的上传和下传函数放在一起，起到数据包流动的时候，相应的日志也在输出。

### 数据包结构

本协议栈中数据包的封装结构参考了TCP/IP协议栈中的数据包封装，数据包在经由应用、传输层、路由层到MAC层向下传输时，每一层中的协议都会在数据包的包头前添加自己的头部控制信息;相反，数据包经由MAC层、路由层、传输层到应用时，每一层协议都会将读取自己层的头部信息，然后将头部信息去除后传给上层。数据包结构如图2-6所示

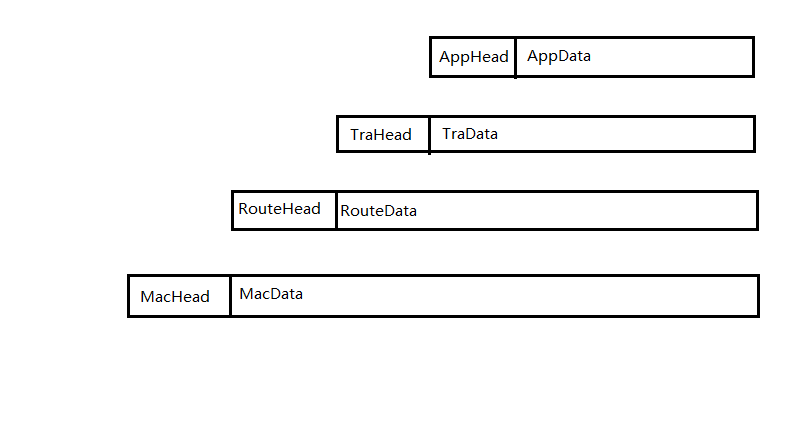


图2-6 数据包结构

数据包由Packet类实现，Packet在内存中的布局如图2-7所示，类中用5个指针指向一段连续内存，start和end分别指向内存的始端和末端，head指向数据包的头部，data指向数据包的内容，tail指向数据包的结尾。

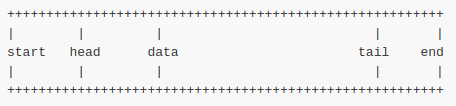


图2-7 数据包在内存中的布局

数据包在协议间传递时，使用同一段的内存，增加效率，只有在以下三种情况中才需要对数据包分配新的内存。

1. 传输层收到来自应用的数据包
2. 物理层收到来自设备上传的数据包
3. 路由层在路由发现阶段发送的hello包

Packet有两种构造方式：

1. Packet p(100);
2. Packet p(buf, 100);

第一种是新申请一段空的内存，大小为100。实际上大小是100+Header大小，Header指所有协议头部的大小。所以当Header是20时，总的大小就是120。

第二种是把已有数据的buf作为数据包，大小是100，这时不会另外为Header申请内存。

Packet使用引用计数的方法，自己控制生命周期。 所以不需要用new来实例化。

template

T\* Header() {return (T\*)head;}

template

void MoveDown() {data=head; head-=sizeof(T)}

template

void MoveUp() {head=data;head+=sizeof(T)}

上面是3个主要的函数，Header是获得数据包的头部，MoveDown是把指针前移，MoveUp是把指针后移。 在接收到上层发来的发送数据请求时，通常就是下面这样添加头部信息：

msg->packet.MoveDown<HeaderType>();

auto h = msg->packet.Header<HeaderType>();

AddHeader(h);

SendDown(msg);

反之收到从下层发来的数据时，通常是下面这样的：

msg->packet.MoveUp<HeaderType>();

auto h = msg->packet.Header<HeaderType>();

GetHeader(h);

SendUp(msg);

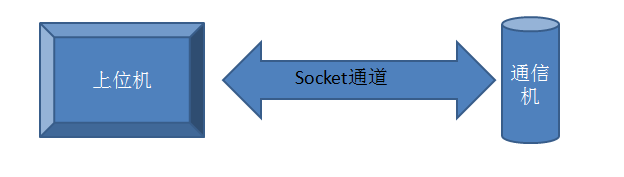
### 设备支持

本网络仿真平台支持串口以及以太网口与真实通信机通信。下面列举使用以太网接口的方法。

要使用指定的以太网接口水声通信机完成HSM协议栈的物理层，必需先理解两个方面的内容——通信机的通信方式以及指令的解释方法啊，在代码中反应的就是需要实现两个个模块：一个负责与通信机通信的工具类，通过这个工具类建立与通信机的通信连接，并与通信机进行读写通信（具体实现在./dev/clientsocket.cc中）；一个是解析通信机发过来的信息的指令解释工具类（具体实现在/phy/clisocket\_sim.cc）；

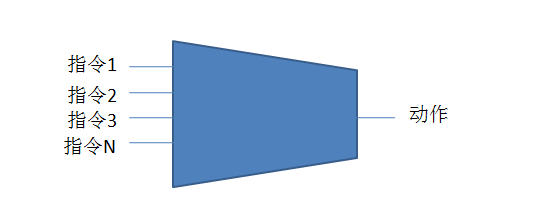
与通信机通信的工具类

由于通信机是采用网线与电脑相连，使用socket来通信的，我们只需要编写一个简单的客户端程序，建立与服务器（通信机）的长连接，之后就在此连接上通信即可。因此在Linux平台下需要学习相关的网络编程知识，例如参考《UNIX网络编程卷一》，客户端的IO模型有很多种实现方式，为了与HSM协议栈进行融合，建议使用协议栈提供的事件驱动库进行网络IO编程。综上所述，需要掌握基本的Linux网络编程知识以及学会使用协议栈中的事件驱动库libev。



解释通信机发来的指令的工具类

这个类的主要作用便是读取数据的前8个字节，分析该指令的作用并执行相应的操作，就像是一个动作的分发器，发起状态机的触发事件，主要的参考资料是通信机的通信序列化协议。



## 协议

### HSM说明

HSM即层次状态机，首先它是一种状态机，可以用来描述网络协议；其次它引入了层次式状态，降低了实现协议的复杂度。

**状态机与协议的关系**

状态机是一种用于表示有限个状态及这些状态间的转移和动作的离散数学模型，可以用一个六元组来定义

其中Σ是输入的字母表（有限非空）；Γ是输出的字母表（有限非空）；是状态集（有限非空）；是初始状态，是的一个元素；δ是状态转移函数），可以用状态图或状态表来描述它；是输出函数。状态机的状态转移和输出由当前状态和输入决定。

状态机在网络协议的设计中起着重大的作用，因为网络协议本身就是一个状态机。如果我们把协议划分出一些状态，把协议内部的原语定义为输入，协议中的处理定义为动作，协议就被抽象为状态、输入、动作、状态转移和处理逻辑的集合，加上初始状态，网络协议就符合状态机六元组的定义。

**层次状态机的特点**

在传统的状态机中，必须为每种可能的参数组合都建立一个状态，这会导致大量的状态和转移，就算是在一个很简单的系统中。层次式状态机，又叫状态图(Statechart)，能够减少这种复杂性和增加状态图的可读性。层次状态机最大的特点就是引入了层次式状态，与面向对象编程语言中的继承类似，一个状态可以继承另外一个状态，如果不对某种输入的动作进行重载，则该状态会执行与父状态一样的动作。可以通过把不同状态对某些输入的动作抽象出来，构造成父状态，对系统进行了逻辑上和实现上的简化，降低系统的复杂度。

**状态、事件、条件和转移**

本协议栈层次状态机的设计主要参考了UML状态图。不过只采用了进入事件(on\_enter)、离开事件(on\_exit)、延迟事件(defer)和守护条件(guard condition)等功能。

* 进入事件是指在进入状态时自动被调用的函数；
* 离开事件则指在离开状态时被调用的函数；
* 延迟事件是指状态可以规定一组在本状态内被延迟的事件。只要本状态一直活跃，这些事件就不会被派发，一旦发生状态转移，这些事件就会重新被派发，好像它们才刚刚到来的样子；
* 守护条件指在状态转移之前先判断此时是否符合某种条件，符合条件时才进行状态转移，否则不转移。

下面是一个简单的例子来说明状态、事件、条件和转移的关系。

如图所示状态图举例，当状态从C转移到F的过程中，进入和离开事件的调用顺序依次是：

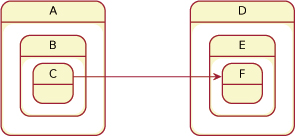


图2.1 进入和离开事件举例

**层次状态机的接口**

本协议栈为用户使用层次状态机提供了简单易用的接口。在协议状态机的六元组中，状态、初始状态、输入、转移函数、动作、处理逻辑都是由用户自己定义的。、

本协议栈还定义了一个状态模板用于帮助用户定义状态，事件模板用于帮助用户定义事件，用户只需要继承之即可（例2.1、例2.2）。

* 状态模板

**template** <**typename** SelfType, **typename** OuterState, **typename** InitState = SelfType>

**struct** **State** : StateBase;

例2.1 状态模板原型

* 事件模板

**template** <**typename** SelfType>

**struct** **Event** : EventBase;

例2.2 事件模板原型

* 对事件作出反应的宏函数。协议栈内定义了多个宏，完成状态转移、延迟事件等功能。其中转移状态的宏会根据输入参数的个数及类型来决定作出不同反应，例如有条件是指转移之前会判断是否满足守护条件，满足条件才转移。有输出是指在状态转移后会执行输出函数（例2.3）。

HSM\_DISCARD(EventType) *// 丢弃事件*

HSM\_DEFFER(EventType) *// 延迟事件*

HSM\_WORK(EventType, Action) *// 执行动作输出*

*// 转移到其它状态，可能会有动作输出，也可能有守护条件*

HSM\_TRANSIT(EventType, State, [,Action] [,Guard])

*// 当不满足守护条件时，延迟事件而不是丢弃*

HSM\_TRANSIT\_DEFER(EventType, State, Guard)

*// 当不满足守护条件时，延迟事件而不是丢弃*

HSM\_TRANSIT\_DEFER(EventType, State, Action, Guard)

例2.3 宏函数

下面是接口使用举例。例2.4 定义了两种事件和一个状态，StateB的父状态为StateA，初始状态是StateC，然后用typedef语句声明了该状态能够处理EventA和EventB两种事件。当EventA到来时，该状态会将之延迟；当EventB到来时，调用函数Guard，根据其返回值判断是否满足守护条件，若返回true，说明满足条件，转移到状态StateD，然后执行函数Action，若返回false，说明不满足条件，将事件丢弃。

在实际使用时，还需要先实例化状态机类，指定初始状态， 才能运行状态机。

**struct** **EventA** : Event<EventA> {};

**struct** **EventB** : Event<EventB> {};

**struct** **StateB** : State<StateB, StateA, StateC> {

**typedef** hsm\_vector<EventA, EventB> reactions;

HSM\_DEFFER(EventA);

HSM\_TRANSIT(EventB, StateD, &Action, &Guard);

};

例2.4 接口使用举例

### 基于HSM的协议实现

下面以UWAloha协议为例，展示如何在本协议栈中实现一个协议。

UWAloha协议是在Aqua-Sim中提出的一个简单的MAC协议。协议的过程可以简单地描述为：发送节点A发送数据，进入等待ACK的状态；接受节点B收到A发来的数据之后，返回一个ACK确认包；节点A在指定时间内收到了ACK确认包，则重新回到空闲状态，否则认为有碰撞发生，进入到退避状态，退避超时后开始重传。

1） 首先将协议抽象为状态、事件、动作以及守护条件。

**状态：**

* + Idle空闲状态：节点空闲
  + WaitACK 等待ACK状态：节点已经发送了数据包，并且正在等待ACK回应
  + Backoff 退避状态：一旦碰撞发送（节点不能收到正确的ACK包），节点将进入该状态。

**事件：**

* + EvSend 发送事件：节点从上层收到发送请求
  + EvRecv 接收事件：节点收到下层传来的包
  + EvTOACK ACK时钟：ACK定时器超时
  + EvTOBACK 退避时钟：退避定时器超时

**动作：**

* + SendUp()：向上层发送包
  + SendDown()：向下层发送包
  + SendACK()：发送ACK应答
  + ReSend()：重传
  + SetACKTimer()：设置ACK定时器
  + SetBACKTimer()：设置退避定时器
  + Defer()：如果节点繁忙，则推迟发送请求
  + Discard()：忽略事件

**守护条件：**

* + ACK：包为ACK应答包
  + DATA：包为数据包
  + ：重传次数大于设定的最大值
  + ：重传次数未超过设定的最大值

2） 画出状态图（如下图所示），其中蓝色部分为路由协议VBF部分，可忽略。

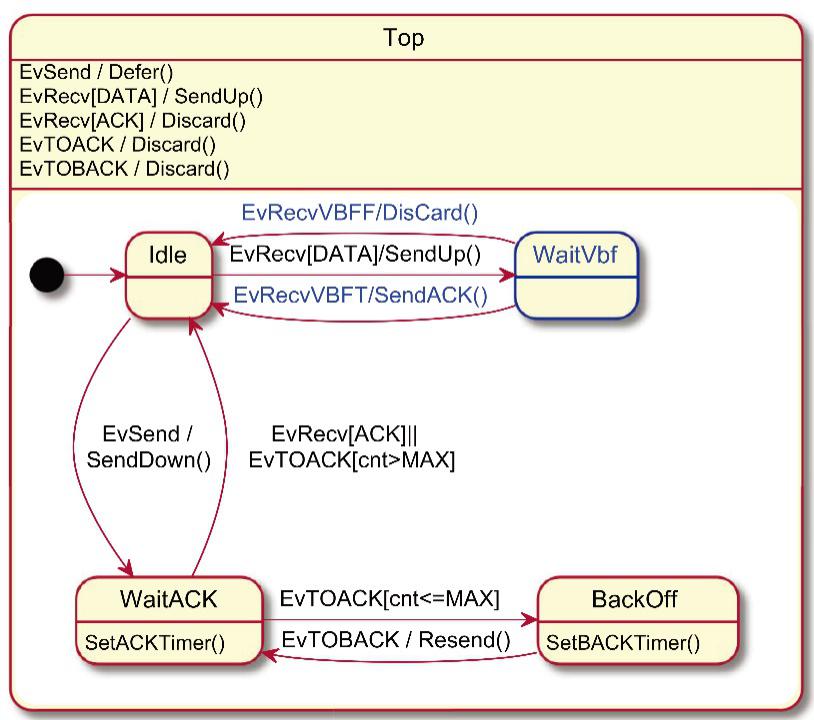


图2.2 UWAloha协议层次状态图

3） 编程完成，基本为以下几个步骤：

* 第一步：定义协议类。这里定义了协议的头部，包括类型标识、源地址和目的地址。

**struct** AlohaHeader *// Define the header*

{

uint8\_t type ;*// packet type*

uint8\_t src ;*// source address*

uint8\_t dst ;*// destination address*

};

*// Declare the protocol class*

**class** **Aloha** : **public** mod :: Module <Aloha , CURRENT\_LAYER ,

CURRENT\_PROTOCOL >

* 第二步：声明事件和状态

*// 声明事件*

*// EvSendReq*

**using** msg :: MsgSendDataReq ;

*// EvRecv*

**using** msg :: MsgRecvDataNtf ;

*// EvTOBACK or EvTOACK*

**using** msg :: MsgTimeOut ;

*// 声明状态*

**struct** Top;

**struct** Idle ;

**struct** WaitAck ;

**struct** BackOff ;

**struct** WaitVbf ;

* 第三步：定义状态，实现协议内部逻辑。以WaitACK状态为例，使用了协议栈内部的宏接口，完成对事件的响应。

**struct** **WaitAck** : hsm :: State < WaitAck , Top > {

HSM\_TRANSIT\_DEFER ( MsgRecvDataNtf , Idle , & Aloha ::

ReceiveAck , & Aloha :: IsAckPacket );}

**struct** **WaitAck** : hsm :: State < WaitAck , Top >

{

**typedef** hsm\_vector < MsgRecvDataNtf , MsgTimeOut > reactions ;

HSM\_TRANSIT\_DEFER ( MsgRecvDataNtf , Idle , & Aloha::

ReceiveAck ,& Aloha :: IsAckPacket );

HSM\_TRANSIT\_TRANSIT ( MsgTimeOut , BackOff , &Aloha ::

HandleTimeouMsg ,& Aloha ::HandleTimeoutAck , Idle );

};

* 第四步：实现处理函数。以传送数据包给上层为例说明。

void Aloha :: SendUp ( **const** Ptr < MsgRecvDataNtf > & ntf)

{

LOG ( INFO ) << " Aloha send up data ";

ntf -> packet . MoveUp < AlohaHeader >();

SendNtf (ntf , UPPER\_LAYER , UPPER\_PROTOCOL );

}

在画出了状态图，并抽取出协议成分之后，步骤1、2、3都是很容易完成的，实现者只需要花时间在步骤4即可。

# 软件使用说明

## 工程目录结构

本程序的工程目录如图3-1所示

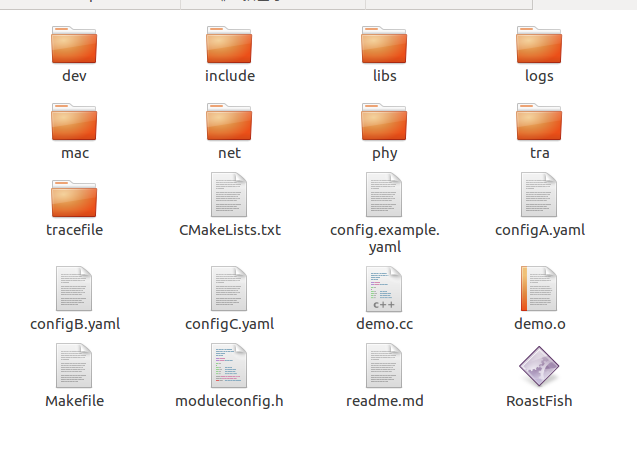


图3-1 协议栈工程目录

各个文件的内容如下：

1. dev文件夹：负责与通信机设备通信的工具类
2. Include文件夹：头文件
3. libs文件夹：使用到库和一些工具类
4. logs文件夹：协议栈运行时的输出日志
5. mac文件夹：mac层协议
6. net文件夹：路由层协议
7. phy文件夹：解析通信机发过来的信息的指令解释工具类
8. tra文件夹：传输层协议
9. tracefile文件夹：协议栈运行时输出的trace文件
10. config.example.yaml / configA.yaml / configB.yaml / configC.yaml：分别为一个配置文件示例和三个不同的配置文件
11. CMakeList.txt / Makefile：编译工程用到的文件
12. demo.cc / demo.o：demo.cc文件包含主函数，.o文件为编译生成文件
13. Readme.md：工程指引文件
14. Roastish：编译生成的可执行文件

## Trace系统

Trace中目前有两个重载函数。

当输出控制包信息时，使用以下函数，各个参数依次是层ID，协议ID，数据包序号，包大小，源节点，上一跳，下一跳，目的节点，hello包序号，mac源节点，mac目的节点，包类型，包状态。

void Trace::Log(int layerID, int protocolID, int packetID, int packetLen, int srcID, int lastID,

int nextID, int destID, int helloSeq, int mSrcID, int mDestID, int ackSeq, string type, string status)

当输出数据包信息时，使用以下函数，各个参数依次是层ID，协议ID，数据包，源节点，上一跳，下一跳，目的节点，hello包序号，mac源节点，mac目的节点，包类型，包状态。

void Trace::Log(int layerID, int protocolID, Packet pkt, int srcID, int lastID, int nextID, int destID, int helloSeq, int mSrcID, int mDestID, int ackSeq, string type, string status)

以上这些参数都需要自己填入，当参数对于本发送或者接收包无意义时，填-1，表示无意义。

那么Trace语句应该放在哪里呢？以下以路由层为例说明。

一、当发送包时，有两种情况。

1.当发送控制包时，应该在填充好头部之后使用Trace的重载函数1，语句如下：

pkt::Packet pkt(0);

DrtPlusHeader \*header = pkt.Header<DrtPlusHeader>();

header->type = HELLO;

header->srcID = full.getNodeID();

header->lastID = helloPktSeq++;

header->nextID = USELESS;

header->destID = BROADCAST;

pkt.Move<DrtPlusHeader>();

Trace::Instance().Log(NET\_LAYER, NET\_PID, -1, -1, (int)header->srcID, -1, -1, (int)header->destID, (int)header->lastID, -1, -1, -1, "hello", "send");

2.当发送数据包时，应该在填充好头部之后使用Trace的重载函数2，语句如下：

if(handleHead(header, dest))

{

//打印log

Trace::Instance().Log(NET\_LAYER, NET\_PID, msg->packet, (int)header->srcID, (int)header->lastID, (int)header->nextID, (int)header->destID, -1, -1, -1, -1, "data", "send");

}

二、当接收到数据包时，先判断是那种包类型，然后使用对应的Trace函数。

1.当判断的是数据包时，使用如下：

if(header->type==DATA)

{

LOG(INFO)<<"header->type==DATA";

Trace::Instance().Log(NET\_LAYER, NET\_PID, msg->packet, (int)header->srcID, (int)header->lastID, (int)header->nextID , (int)header->destID, -1, -1, -1, -1, "data", "recv");

}

2.当判断的是控制包时，使用如下：

else if(header->type==HELLO){

LOG(INFO)<<"header->type==HELLO";

Trace::Instance().Log(NET\_LAYER, NET\_PID, -1, -1, (int)header->srcID, -1, -1, (int)header->destID, (int)header->lastID, -1, -1, -1, "hello", "recv");

}

## 增加协议

用户如果需要自己增加自定义协议，可参考以下步骤

1. 按照2.6节层次状态机的方法将协议实现，并将协议文件保存在相应的工程文件夹中
2. 在Makefile中增加该协议
3. 在moduleconfig.h和configX.yaml配置文件中增加该协议需要的参数
4. Make clean && make
5. 运行测试

## 标准输入测试和循环测试

本协议栈有两种运行模式，分别为标准输入测试和循环测试。标准输入测试一般用于验证协议是否可以正常在协议栈中正常运行，例如：是否可以跟上下层协议正常沟通，协议运行过程是否会出现程序崩溃，协议运行结果是否跟预期一致等等。在验证了协议可以正确运行之后，可以根据需要进行循环测试，循环测试一般用于测试协议的性能好坏，例如可以让协议栈运行持续发送数据包，收集trace文件，利用trace文件进行协议的性能分析。

**标准输入测试**

本协议栈可以在终端输入信息发送到目的节点，具体运行方式是在终端输入“ ./RoastFish configX.yaml”，等待节点初始化完成，如果使用的是动态路由协议则需要额外等待路由发现过程结束，之后在终端输入字符串并回车，终端输入的格式是“目的节点ID+数据”，如“1abcdefg”，表示向目的节点1发送“abcdefg”。即完成了向目标节点1发送字符串的操作。

**循环测试**

使用循环测试需要在配置文件中将循环测试模式打开并将所需要的参数配置好即可，循环测试使用了事件驱动库里面的定时器功能。具体配置参数见图3-2

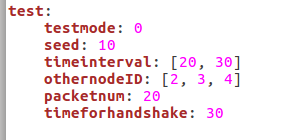


图3-2 循环测试配置文件

循环测试的参数说明如下

1. testmode：是否打开测试模式，0为关闭测试模式，1为打开测试模式
2. seed：随机数种子
3. timeinterval：发送数据包的时间间隔，[20, 30]表示两个发送的数据包之间的间隔为20～30秒之间的随机数，随机数种子由seed提供
4. othernodeID：发送数据包的候选目标节点，[2, 3, 4]表示在每次发送时都会从2, 3, 4中选择一个作为数据包的目标ID
5. packetnum：循环测试发送数据包的个数，20表示协议栈在发送完20个数据包后停止发送。
6. timeforhandshake：程序启动后开始测试模式的等待时间，在程序启动后，节点需要进行初始化，在某些情况下需要等待其他节点开始并初始化，在使用了动态路由协议时需要等待路由发现过程结束，因此本参数的值应该是以上三个等待时间的组合，视具体情况需要设置不同的等待时间，单位是秒。

在配置文件中完成了参数配置后，就可以开始进行循环测试，具体是在终端中运行“ ./RoastFish configX.yaml”，等待循环测试结束后即可去trace文件夹中将本测测试产生的trace文件取出进行协议的性能分析。

## Libev

### Libev的安装

1. 进入http://dist.schmorp.de/libev/下载最新版本，链接下载的是4.24版本。
2. 解压在任意目录上。
3. 进入目录运行后：

sh autogen.sh

./configure && make

sudo make install

执行第一句若出现/autogen.sh: 4: autoreconf: not found，就安装一些组件：sudo apt-get install autoconf automake libtool

1. 检查是否成功安装

ls -al /usr/local/lib |grep libev (若你没有指定安装目录，则默认是/usr/local/lib 路径)

1. 在官方文档http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod中学习使用libev库。

### Libev在协议中的使用

本协议栈需要按需处理IO事件以及定时器事件，还需要利用prepare机制来执行调度器代码。

1. IO事件处理

其中，IO事件的处理如下：

ev\_io m\_read\_listen;

ev\_io\_init(&m\_read\_listen, read\_listen, listenfd, EV\_READ);

ev\_io\_start(LibevTool::Instance().GetLoop(), &m\_read\_listen);

第一句是声明IO事件的句柄，第二句是设置IO事件的回调函数，关心的文件描述符以及事件，第三句是开启该IO事件的侦听器。

1. 定时器事件处理

定时器事件通常是提供给协议使用，因此已经封装好提供给层次状态机使用，通过函数SetTimer(时间值, 定时器ID) 设置定时器时间并开启定时器，定时时间一到，层次状态机会发起一个msg::MsgTimeOut事件，状态机捕捉该事件即可。

在协议外如何定义一个新的定时器事件呢？和IO事件类似：

ev\_timer m\_test\_timer;

ev\_timer\_init (&m\_test\_timer, testsendpacket, timeforhandshake, 1.0);

ev\_timer\_start (LibevTool::Instance().GetLoop(), &m\_test\_timer);

第一句是声明定时器事件的句柄，第二句是设置定时器事件的回调函数，设定定时时间，以及设定是否循环启动，第三句是开启该定时器事件的侦听器。

1. Prepare机制

Libev的prepare是让你在每次事件循环之前都调用一段代码，在本平台下，主要是启动调度器的代码。

ev\_prepare prepare\_watcher;

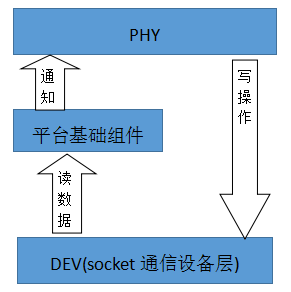
ev\_prepare\_init(&prepare\_watcher, sched::Scheduler::Sched);

ev\_prepare\_start(LibevTool::Instance().GetLoop(), &prepare\_watcher);

和上面设置事件一样，先定义句柄，再设置回调函数，最后开启事件侦听器即可。

## Socket通信设备工具类

常用的设备是socket通信设备，下面详细介绍一下该工具类的设计，该类是提供给PHY层使用的，是PHY层的一个成员变量，PHY层通过调用该成员变量的方法将数据发送给真实通信机。而从真实通信机接收的数据，则由本平台的dap等组件负责通知PHY层，大致关系如下图所示。



Dev层的设计主要是集合事件驱动库的IO事件来实现通过socket与通信机交互数据的，例如在初始化函数中与通信机连接之后，执行下面代码：

ev\_io\_init(&read\_io\_, ReadCB, fd\_, EV\_READ);

ev\_io\_start(LibevTool::Instance().GetLoop(), &read\_io\_);

ev\_io\_init(&write\_io\_, LibevTool::WritefdCB<WriteQueue>, fd\_, EV\_WRITE);

那么就可以接收数据后，自动回调ReadCB函数，向本平台的基础组件中写入数据，dap会负责将数据传递给PHY层，PHY层再传递给MAC层。

而发送数据也是类似的操作，自动回调LibevTool::WritefdCB<WriteQueue>函数，该函数会从WriteQueue中获取数据并通过socket发往通信设备中。

# 已实现的协议

## 路由层

### 数据包的格式

首先讲一讲协议栈中数据包的格式，如下图所示：

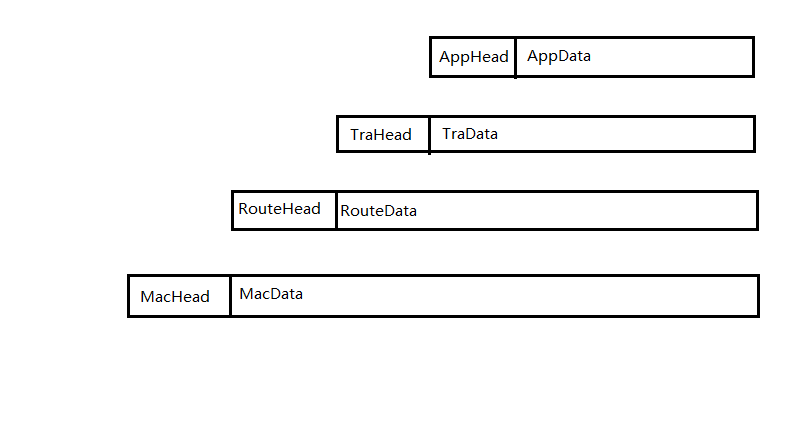


图1.Stack数据包

其中，AppData就是用户输入的数据，AppHead就是应用层的头部。TraData就是AppData+Appdata，TraHead就是传输层的头部。Udp协议头部：由于无用，已经无效（0字节）RouteData就是TraData+TraData，RouteHead就是路由层的头部。

### 静态路由协议

**协议简述**

静态路由协议就是根据网络拓扑，人为的设定每个节点的路由表。

**实现说明**

Srt协议头部：源地址，目的地址，下一跳地址

|  |  |  |
| --- | --- | --- |
| SrcID（1字节） | destID（1） | nextID（1） |

Drt协议头部：包类型、源地址，目的地址，下一跳地址

|  |  |  |  |
| --- | --- | --- | --- |
| Type（1） | srcID（1） | destID（1） | nextID（1） |

DrtPlus协议头部：包类型、源地址，目的地址，本跳地址，下一跳地址

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type（1） | srcID（1） | destID（1） | lastID（1） | nextID（1） |

下面说明一下基于路由表的路由协议怎么传送数据包。如图4-1所示，

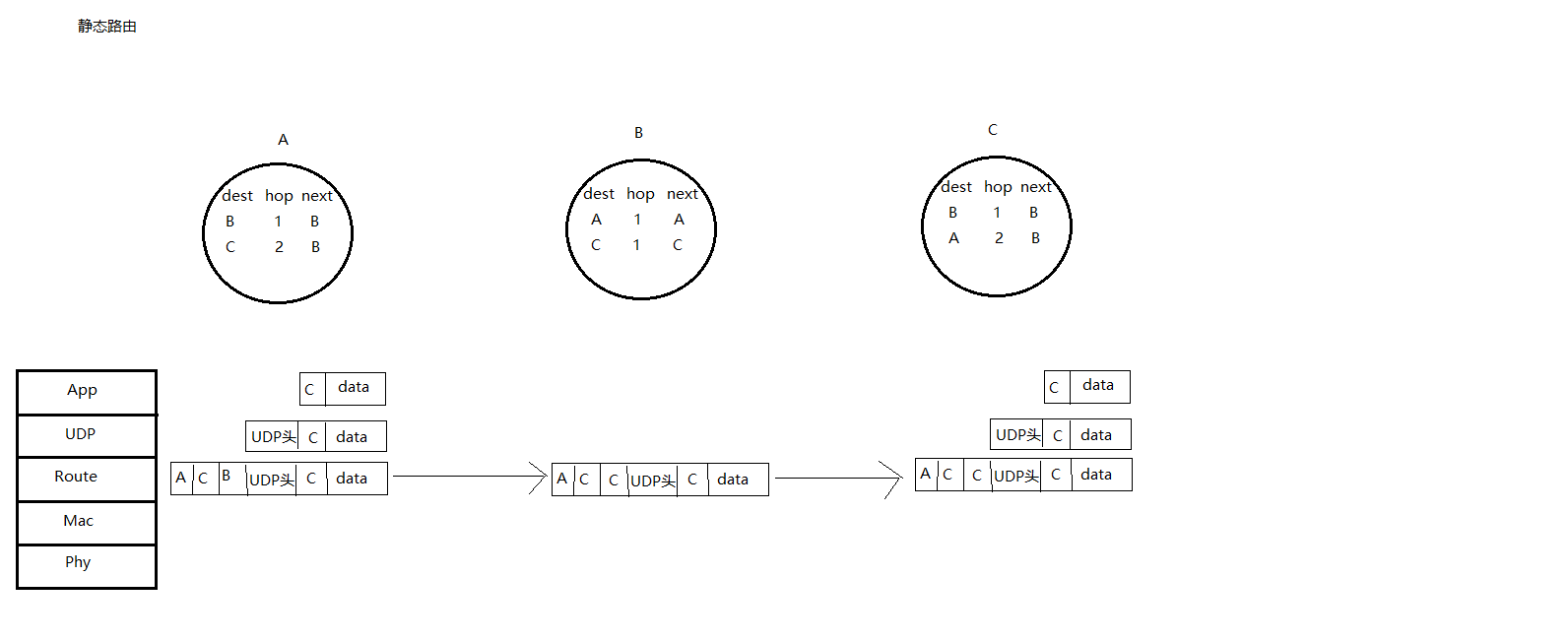


图 4-1 静态路由协议

如上图所示，从路由表可以知道，当一个节点要发送数据包给某个节点时（dest），需要几跳（hop），下一跳要传送给谁（next）。一下以A->C为例说明一下传送过程。

1. .首先数据包经过层层封装到达路由层。
2. 路由层根据目的地址信息查询路由表得到下一跳，将A，C，B分别填入路由层头部，其中A的意思是原地址，C的意思是目的地址，B的意思是下一跳地址。
3. 填完路由层头部之后，经过下层的层层封装之后，通信机将数据包广播出去。
4. B通信机收到了数据包，层层解包之后到达路由层，发现路由层头部的信息为A，C，B，得知自己正是一个转发节点，也知道目的地址是C，查询路由表后，将下一跳由B改成C，然后往下传，经过下层的层层封装之后，通信机将数据包广播出去。
5. 之后C通信机收到了数据包，层层解包之后到达路由层，发现路由层头部的信息为A，C，C，得知自己就是目的节点，于是路由层将数据包往传输层传送。

### 动态路由协议

**协议简述**

动态路由协议的数据包传送也是依据路由表的，和静态路由传送过程一样。但是静态路由的路由表是由认为设定的。而动态路由的路由表是由节点动态发现的。

**实现说明**

路由发现过程如图4-2：

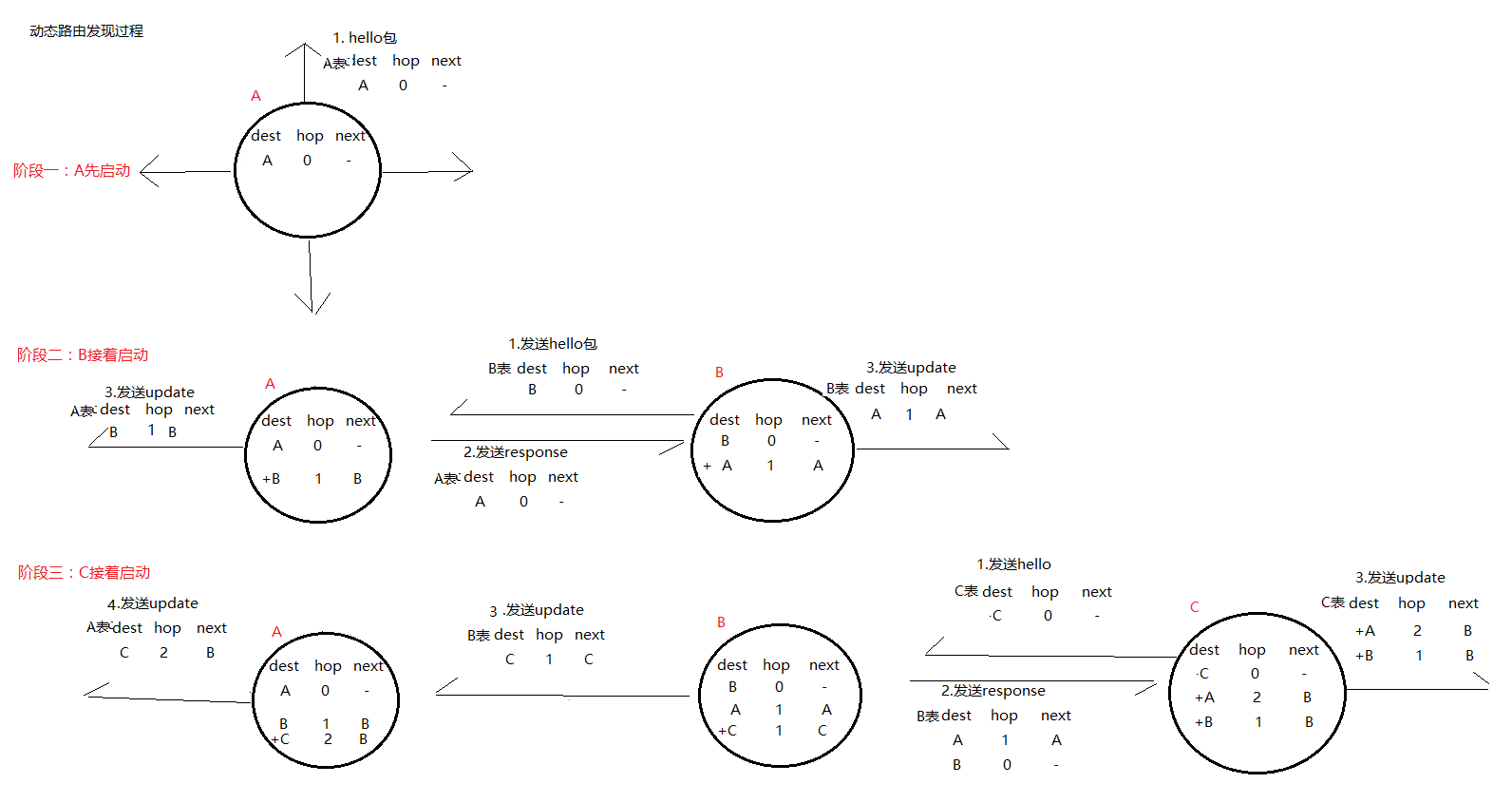


图 4-2 动态路由发现过程

1. 第一阶段：

A启动后在自己的路由表中加入A 0 -的路由信息，并将该路由信息填入hello包发送。

等待一段时间后没有收到response包，那么路由表就不变了。

1. 第二阶段：

B接着启动，在自己的路由表中加入B 0 -的路由信息，并将该路由信息填入hello包中发送。A收到B的hello包后在自己的路由表中加入B 1 B的路由信息，并且将自己路由表中的所有信息填入response包中发送。B收到response包后，将A发送后来的路由表信息填入路由表。

因为A和B的路由表都更新了，所以A和B都会将更新的内容填入update包并发送，但显然其周围并没有节点，所以接下来不发生变化了。

1. 第三阶段：

C接着启动了，在自己的路由表中加入C 0 -的路由信息，并将该路由信息填入hello包中发送。B收到C的hello包之后，在自己的路由表中加入B 1 B的路由信息，并将自己路由表中的所有信息填入response中发送，C收到response包后，在自己的路由表中加入了两条路由信息。因为B和C的路由表都有更新，所以B和C都会将更新的内容填入update包并发送，但是C的update包并不能带来其他节点路由表的变化。而B的update包被A收到后，A就在路由表中加入了C 2 B。因为A的路由表有更新，所以发送了更新包，但是对其他节点的路由表并不造成影响。

## MAC层

### Simple\_aloha:对所有数据包都不要求重发

**协议简述**

Simple\_aloha是一个简单的aloha协议，对收到物理层的数据包，都上传至路由层，对路由层下发的数据包，都下发给物理层，不做其他处理

**实现说明**

本协议数据包头部有三个字段，分别为数据包类型、数据包目的MAC地址和数据包源MAC地址。

struct SimpleAlohaHeader

{

uint8\_t type;

uint8\_t src;

uint8\_t dst;

};

协议的实现状态图如图4-3所示，由于协议比较简单，只需要两个状态即可。

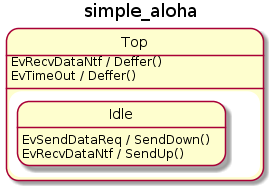


图4-3 simple\_aloha状态图

由状态图可以看出，协议中定义了两个个函数，具体功能如下

1. SendDown（）：当协议收到来自路由层的数据包时，首先将MAC层的头部信息填充，包括数据包类型、源MAC地址，数据包序号和目的MAC地址。
   * 1. 将数据包下发，不进行其他操作
2. SendUp（）：当协议收到来自物理层的数据包时，将数据包上传至路由层，不进行其他操作。

### New\_aloha:对除广播包之外的每个数据包都要求重发

**协议简述**

New\_aloha是一个对aloha协议增加了重发功能的协议。对发送出的数据包除了广播包之外都要求重发

1. 在发送方，对于下发的数据包，检查其类型，若为非广播包或ACK，则将数据包下发，将该数据包保存在重发数据包列表中，并设置重传超时计时器;若为广播包或ACK，则将数据包下发，不做其他处理
2. 在接收方，检查收到的每个数据包，若为ACK，则在重发数据包列表中找到该数据包删除，若为广播包，则将其上传则路由层，若为数据包，则将其上传至路由层，并发送ACK

**实现说明**

本协议数据包头部有四个字段，分别为数据包类型、数据包目的MAC地址、数据包源MAC地址和数据包序号。

struct NewAlohaHeader{

uint8\_t type;

uint8\_t destID;

uint8\_t sourID;

uint16\_t serialNum;

};

协议的实现状态图如图4-4所示，包含三个状态，顶层状态、空闲状态和等待响应状态，状态之间的转移很简单，具体的逻辑实现在函数中体现。

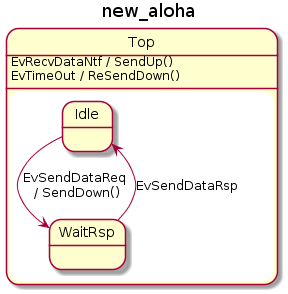


图4-4 new\_aloha状态图

由状态图可以看出，协议中定义了三个的函数，具体功能如下

1. SendUp（）：当协议收到来自物理层的数据包时，首先读取头部的包类型信息
   * 1. 若该包为数据包，且目的地址为自己，则将数据包上传至路由层并发送ACK
     2. 若该包为广播包，则将该包上传至路由层，不做其他操作
     3. 若该包为ACK，则在待重发数据包列表中找到该数据包，将其删除
2. SendDown（）：当协议收到来自路由层的数据包时，首先将MAC层的头部信息填充，包括源MAC地址，数据包序号，目的MAC地址，数据包类型
   * 1. 若该包为广播包，则将其下发至物理层，不进行其他操作
     2. 若该包为数据包，则将其下发至物理层，设置超时重传定时器，并将该数据包存至待重发数据包列表中
3. ReSendDown（）：当协议收到超时事件时，则在待重发数据包列表中找到相应数据包
   * 1. 若该数据包的剩余重发次数为0，则重发数据包，并将该数据包从待重发列表中删除
     2. 若该数据包的剩余重发次数大于零，则重发该数据包，将其重发次数减一，重新设置超时计时器

### Aloha\_special : 对部分重要数据包进行CRC校验并要求重发

**协议简述**

Aloha\_special是一个对aloha协议进行部分修改的协议。修改的部分将数据包分为包含重要信息的数据包与普通数据包，对于包含重要信息的数据包进行CRC校验并增加重传机制。具体修改如下：

1. 在发送方增加了对部分包含重要信息的数据包进行32位的CRC校验，并要求该接收方收到该数据包后回复ACK，若在超时时间内没有收到ACK，则会进行超时重传。
2. 在接收方收到包含重要信息的数据包时，则会对该数据包进行CRC校验，若校验成功，则回复ACK;校验失败则不做处理，等待重传。
3. 对于不包含重要信息的数据包，则没有CRC校验，没有重传机制。

**实现说明**

本协议数据包头部有四个字段，分别为数据包类型、数据包目的MAC地址、数据包源MAC地址和数据包序号。

struct AlohaSpecialHeader{

uint8\_t type;

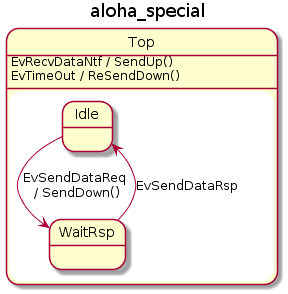
uint8\_t destID;

uint8\_t sourID;

uint16\_t serialNum;

};

协议的实现状态图如图4-5所示，包含三个状态，状态之间的转移很简单，具体的逻辑实现在函数中体现。

****

**图4-5 aloha-special状态图**

由状态图可以看出，协议中定义了三个的函数，具体功能如下

1. SendUp（）：当协议收到来自物理层的数据包时，首先读取头部的包类型信息，
   * 1. 若数据包为重要数据包，并且目的地址为自己，则进行CRC校验，若CRC校验成功，则回复ACK并将数据包上传至路由层，若CRC校验失败，则不做处理等待重发。
     2. 若数据包为ACK类型并且目的地址为自己，则读取包的序号和源地址，在待重发的列表中将重发计时器取消。
     3. 若数据包为普通数据包且目的地址为自己或广播包，则将数据包上传至路由层，不做其他处理
2. SendDown（）：当协议收到来自路由层的数据包时，首先将MAC层的头部信息填充，包括源MAC地址，数据包序号，目的MAC地址，数据包类型
   * 1. 若数据包为重要数据包，则将数据包加上CRC校验字段，将该包下发给物理层，将该数据包加入待重发列表，并设置超时重传定时器。
     2. 若数据包为广播包或普通数据包，则将该数据包下发给物理层，不做其他处理。
3. ReSendDown（）：当定时器超时时，在超时列表中寻找对应的数据包
   * 1. 若该数据包的剩余重发次数为0,则重发数据包，并将该数据包从待重发列表中删除
     2. 若该数据包的剩余重发次数大于零，则重发该数据包，将其重发次数减一，重新设置超时计时器

另外本协议还有一个实现CRC校验的函数GenCRCCheckSum（），功能为生成CRC校验码。