
基于视频服务 QoE 预测的 轻量级流量特征器的设计与实现

摘要

目前移动互联网的发展迅速，Internet 上视频流量的规模日益提升。对于视频服务提供商和网络供应商，保证用户体验质量（Quality of user Experience, QoE）是提高用户粘性的重要手段。而通过 QoE 预测可以帮助服务供应商保证 QoE。目前，QoE 预测主要通过流量分类技术实现。流量分类首先需使用特征工程方法获取流量特征，随后利用获取到的特征进行分类预测。传统的流量分类方法基于数据包的传输层端口号和有效载荷中的内容进行特征提取。然而随着网络加密技术的发展和普及，如今互联网中绝大多数网络流量均已使用各式加密算法进行加密，传统的特征工程方法难以保证特征的有效性。因此设计可提取加密流量特征的特征器对于流量分类技术在如今的发展和应用有重要意义。

随着机器学习技术在各个领域的普及，加密流量分类与机器学习算法的结合使得视频 QoE 预测技术产生新的突破。在特征工程方面有机器学习和人工设计两种方法。前者利用机器学习算法将特征工程与分类模型训练整合。优点是可由算法评估特征的有效性进而动态调整特征，使其可以适用于不同网络环境。缺点是运算复杂度高且无法进行实时流量特征提取。后者的优点是特征提取效率高，但也有适用性差、受网络环境影响大的缺陷。综合考虑以上问题，本研究考虑对在当今网络环境中占主导地位的视频流量进行特征提取，设计并实现了面向视频服务 QoE 预测进行特征提取的轻量级流量特征器，可对视频流量进行识别并提取其流量特征。

本研究分为三个阶段，首先完成对视频服务 QoE 指标的调研，研究流量特征与 QoE 指标间的关联。多角度，多协议层次地设计面向加密视频服务 QoE 预测的流量特征集；其次针对轻量级的设计要求，设计高效率的特征计算方法与低内存占用的特征存储方法，进而实现轻量级流量特征器。最后将实特征器部署于开源路由器 OpenWrt 中，通过对现实视频流量的特征提取，分类识别与 QoE 预测验证了特征集合与特征器的可靠性，可以为视频 QoE 预测提供有效的数据支撑。

关键词 流量分类 流量特征 视频服务 QoE 预测 OpenWrt

Design And Implementation Of Lightweight Flow Feature Generator Based On Video Service QoE Prediction

ABSTRACT

Today, the scale of video traffic on the Internet has reached 82% of all traffic. For any video service provider, ensuring the Quality of User Experience (QoE) is an important means to improve user stickiness. And through QoE prediction can help service providers to ensure QoE. At present, QoE prediction is mainly realized through traffic classification technology. Traffic classification refers to classifying and identifying network traffic using a classification algorithm, which consists of two steps. First, we need to use feature engineering to obtain traffic features, and then use the obtained features for classification prediction. With the development and popularization of network encryption technology, the vast majority of network traffic on the Internet has been encrypted with various encryption algorithms, and traditional feature engineering methods are difficult to ensure the validity of features. Therefore, designing a signature that can extract encrypted traffic features is of great significance for the development and application of traffic classification technology today.

There are currently two feature engineering methods for encrypted traffic classification, machine learning and manual design. The former utilizes machine learning algorithms to integrate feature engineering with classification model training. The advantage is that the effectiveness of the features can be evaluated by the algorithm, and then the features can be dynamically adjusted to make them suitable for different network environments. The disadvantage is that the computational complexity is high, and real-time traffic feature extraction cannot be performed. The advantage of the latter is that the feature extraction efficiency is high, but due to the complexity of the network environment and the limitations of artificial design, this method also has shortcomings such as poor applicability and great influence by the network environment.

Taking the above problems into consideration, this study considers feature extraction for the dominant video traffic in today's network environment, and designs and implements a lightweight traffic feature for feature extraction for video service QoE prediction, which can identify video traffic. and extract its traffic characteristics. This research is divided into three stages. First, the survey of video service QoE indicators is completed, and the correlation between traffic characteristics and QoE indicators is studied. Design the traffic feature set for QoE prediction of encrypted video services from multiple perspectives and multi-protocol

levels; secondly, according to the lightweight design requirements, design high-efficiency feature calculation methods and low-memory-occupancy feature storage methods to achieve lightweight traffic. feature. Finally, the real feature device is deployed in the open source router OpenWrt, and the reliability of the feature set and the feature device is verified by feature extraction, classification identification and QoE prediction of real video traffic, which can provide effective data support for video QoE prediction.

KEY WORDS flow classification flow feature video service QoE prediction
OpenWrt

目 录

第一章 绪论	1
1.1 选题背景与意义.....	1
1.2 课题研究内容.....	2
1.3 论文的组织与结构.....	3
第二章 相关技术介绍	4
2.1 数据包获取技术介绍.....	4
2.1.1 数据包获取函数库介绍.....	4
2.1.2 数据包捕获程序原理.....	4
2.2 开源路由器 OpenWrt 介绍.....	5
2.2.1 OpenWrt 概述.....	5
2.2.2 OpenWrt 的网络适配器配置.....	5
2.3 加密流量分类技术最新研究进展.....	8
2.3.1 视频流量预测方法.....	8
2.3.2 QoE 预测方法.....	8
2.4 本章小结.....	8
第三章 系统需求分析	9
3.1 系统总体需求.....	9
3.2 系统功能需求.....	10
3.2.1 数据包获取功能.....	10
3.2.2 包信息采集功能.....	10
3.2.3 流量控制功能.....	10
3.2.4 流量特征生成功能.....	10
3.2 非功能需求分析.....	11
3.3 本章小结.....	11
第四章 概要设计	12
4.1 系统结构设计.....	12
4.2 接口设计.....	12

4.3 概要设计	13
4.3.1 数据包获取层	13
4.3.2 信息处理层	13
4.3.3 流量管理层	14
4.3.4 特征计算层	15
4.4 数据流程图	15
4.5 本章小结	15
第五章 系统详细设计与实现	16
5.1 数据包获取层的设计与实现	16
5.1.1 libpcap 库函数的详细说明	16
5.1.2 数据包获取模块设计	19
5.2 信息处理层的设计与实现	19
5.3 流量管理层的设计与实现	23
5.4 特征计算层的设计与实现	26
5.4.1 特征存储设计	26
5.4.2 特征计算设计	27
5.5 本章小结	30
第六章 软件部署与测试	31
6.1 软件部署	31
6.1.1 宿主机交叉编译环境配置	31
6.1.2 虚拟机动态库配置	31
6.1.3 宿主机动态库的交叉编译	32
6.1.4 源码交叉编译与移植	33
6.1.5 错误示例	33
6.2 实验环境选择	34
6.3 软件测试	35
6.3.1 离线流量特征器的测试	35
6.3.2 实时流量特征器的测试	36
6.4 数据分析	38

6.5 本章小结.....	43
第七章 总结与展望.....	44
7.1 总结.....	44
7.1.1 系统实现的意义.....	44
7.1.2 系统已完成的工作.....	44
7.1.3 系统的改进方向.....	45
7.2 展望.....	45
7.3 心得与收获	45

第一章 绪论

1.1 选题背景与意义

随着移动互联网的发展，短视频业务正迅速扩张，各类流媒体流量已占据互联网上全体网络流量的 82%。对于视频服务的提供者，使用流量分类技术进行 QoE 预测来保证 QoE，是提高用户粘性的重要手段。流量分类技术包含特征工程和分类模型训练两步。其中特征工程指从收集到的网络流量中提取特征。传统特征工程方法将传输层端口号或数据包的有效载荷作为决定性特征。前者称端口号分类法，而后者被称为深度包检测 (DPI)。如今，随着加密技术在用户隐私保护方面的广泛应用，加密流量在网络流量中的占比急剧上升，这使得应用层负载不再能作为特征提取的依据。而端口号分类法也因端口跳变技术和端口伪装技术的出现而丧失准确性。因此，设计加密流量分类的方法成为了加密流量分类方面的研究重点。

将机器学习用于加密流量分类是当前热点之一，如 FS-NET^[1]是一种将特征工程与模型训练整合的端到端模型，在特征提取和分类模型训练间部署反馈神经网络，由机器学习算法进行流量特征的选取与动态调整。此外，仅将机器学习应用于分类预测，保持人工设计特征集得方案同样有重要意义。Moore 等^[2]人设计了 248 种流量特征，并生成带流量标签的流量数据集。他们的特征设计方案为后续各式加密流量分类相关研究提供了指导意义，且他们的流量数据集也为后续研究提供了丰富的数据。但是上述两种方案均无法实现实时提取，后者是因为所设计的特征往往需要获取整条流量的数据，前者则为效率所限制。进一步的，在视频服务的 QoE 预测方面，目前的主流观点是从启动延迟，分辨率，停滞三个角度评判 QoE 指标。Mazhar M H, Shafiq Z 等^[3]人设计了一套包含网络层和传输层特征的特征集合对上述三种指标进行预测。该研究的特征选取基于网络供应商可以识别的网络信息，包括视频提供者的 IP，DNS 响应，TLS 握手中的 SNI 信息以及网络层和传输层首部。Bronzino F, Schmitt P 等^[4]则在上述研究的基础上进一步阐述了各个 QoE 指标的内在意义，并针对启动延迟和分辨率设计了更具代表性的特征。上述两研究生成的特征均由机器学习分类算法生成了准确率较高的 QoE 预测结果。但其结果也反映出不同视频网站的视频流量特征有差异，会导致预测出现偏差的问题。

进行加密流量特征提取，需要针对加密协议特点进行有针对性的特征集设计。目前流行的加密协议主要有位于应用层的 HTTPS，位于传输层的 TLS 与 QUIC，位于网络层的 IPsec 等^[5]。在不同协议间，相同的流量特征可能反映着截然不同的网络特点。因此，流量特征提取器需要具有识别加密协议类型的能力。而仅凭加密后的报文完成对加密协议

的准确判断是困难、且资源消耗大的。本研究着眼于当前 Internet 上占主导地位的视频流量。据统计，网络上 82% 的流量是视频流量。由于视频服务有较高的实时性、准确性要求，它们大多采用 HTTPS 或 QUIC 协议进行加密。其中，HTTPS 是综合了对称加密和非对称加密的 HTTP，其效率由 HTTP 协议保证；而 QUIC 基于 UDP 自定义的连接，重试，多路复用，流量控制等功能保证效率。HTTPS 基于传输层的 TCP 协议，而 QUIC 则是在传输层 UDP 协议的基础上改进而得的。因此仅凭传输层协议类型即可区分这两种加密协议，进而更加有效地提取特征。

针对上述问题，本研究整合已有研究结果，旨在设计轻量级的，面向加密视频服务 QoE 预测的流量特征器；研究视频流量的流量特征与视频服务 QoE 之间的关联，并由此设计特征集，完成轻量级流量特征器的设计与实现；并在路由器系统上完成部署与运行。

1.2 课题研究内容

本研究从视频服务的 QoE 预测角度出发，研究视频流量的流量特征与视频服务 QoE 之间的关联，并由此设计特征集，完成轻量级流量特征器的设计与实现。并在路由器系统上完成部署与运行。本研究主要内容有以下几点：

1) 特征器轻量级目标的实现方式：

流量特征器最终需要部署于路由器。本研究选取四大开源路由器项目之一的 OpenWrt 路由器作为目标系统进行设计。该系统基于 linux 内核，提供多种软件包与动态库，包括本项目所必须的数据包捕获动态库，且提供对 C++ 可执行程序的支持。但该路由器系统仅有 16M 内核空间与 128M 磁盘空间，因此实现轻量级目标是本研究的必然要求。在软件设计方面，本研究设计了高效的特征计算与特征存储方案，以降低 CPU 和内存的占用。在代码实现方面，由于路由器系统没有配置易于编码、调试的图形化界面。本研究选择在与路由器系统同样基于 Linux 操作系统的 Ubuntu 中进行编码调试。但由于系统指令集架构、数据存储方式（大/小端）的不同，在 Ubuntu 中编译链接得到的可执行程序无法直接在路由器中运行。因此本研究进行了交叉编译环境的配置，完成了跨平台运行与部署。

2) 面向 QoE 预测的加密视频流量特征设计：

视频服务的流量大多使用 HTTPS 或 QUIC 协议进行加密，其特点是无法利用有效载荷，但可以分析传输层至数据链路层的首部信息。此外，视频流量的另一个显著特点是一旦服务开始并正常运转，上行流量将只剩下段请求报文和确认报文。而这两种报文可以利用报文长度进行区分，由此可以将报文划分成不同的段（segment），进而间接推断应用层信息。由 Bronzino F, Schmitt P^[5]等人的研究可知，启动延迟即为用户视频缓冲区装填的过程，因此视频段下载越快，启动延迟越短。与启动延迟相关的特征有：段

大小和段请求间隔。预测分辨率最重要的特征同样是段大小，此外包计数，包字节计数，每秒包数同样可作为预测分辨率的指标。

在综合上述研究结果及 Mazhar M H, Shafiq Z^[4]等人关于网络层、传输层流量特征的设计后，本研究最终选取：包长、包到达间隔、TCP 窗口大小、段大小、段请求间隔、段持续时间的不同统计数据；以及流持续时间、平均比特率，平均每秒包数、重传次数、TCP 紧急指针计数、TCP 协议报文数量占比作为轻量级特征器提取的特征集。

3) 高效特征计算与存储方案：

对于流量特征的统计数据，传统特征器采取在收集一条完整流量的信息后进行统计计算的方法。该方法需存储一条流量所有特征的数据，而路由器系统无法提供足够的磁盘空间，将导致系统崩溃。本研究基于增量计算的方法设计增量特征，以较低的计算量实时更新可用于计算特征统计数据的变量，如个数，线性和，平方和。且对于每个特征，仅需存储上述三个变量，大大减少了特征计算的运算量和特征存储的空间占用。此外，本研究设计了衰减方案，使得每次特征输出都聚焦于最新数据包和最新链路状态。

1.3 论文的组织与结构

第一章 绪论，介绍本研究的选题背景及意义。简要说明研究内容并介绍论文整体结构。

第二章 技术综论，详细介绍本研究所用到的关键技术。

第三章 系统需求分析，描述流量特征器的系统需求分析。

第四章 系统概要设计，介绍流量特征器的概要设计。

第五章 系统详细设计与实现，详细介绍本研究设计的流量特征器中各个模块的功能，及代码的具体实现。

第六章 软件部署与测试，对软件的功能进行测试。

第七章 总结与展望，对本研究的内容进行总结，支持其不足及改进方向。最后对相关技术的发展和本研究对未来的意义做出展望。

第二章 相关技术介绍

2.1 数据包获取技术介绍

2.1.1 数据包获取函数库介绍

libpcap（数据包获取函数库）^[6]是 unix/Linux 平台下的数据包捕获函数库，可以在各种类 unix 平台下工作。许多网络监控软件以 libpcap 为基础，如 Linux 下的 tcpdump。其主要功能有：数据包捕获、数据包过滤、数据包分析、数据包存储。libpcap 主要由两部分组成，包括网络分接头与数据过滤器。其中网络分接头基于旁路机制从网络适配器中获取数据包信息，之后将数据包发送给数据过滤器判断是否满足过滤条件。满足条件的数据包可利用函数库创建的 PF_PACKET 套接字从二层驱动中获取。由此在用户空间实现了系统无关的 API 接口。为实现透明抓包——即数据包采集或丢弃不影响流量本身的传播，libpcap 的网络分接头采用旁路机制在内核内存中过滤和缓冲数据包，并直接发送给调用 libpcap 的应用程序。抓包的实现流程如下：

- 1) 网卡接收到数据包
- 2) 网卡进行 DMA 传送，将数据包由网卡寄存器发送至内核缓冲区 ring buffer。
- 3) 网卡唤醒处理器
- 4) 驱动程序从 ring buffer 将数据包拷贝至内核专用数据结构 skbuff
- 5) 过滤器读取 skbuff，将符合过滤条件的数据包拷贝至内核缓存
- 6) libpcap 用户空间 API 直接调用套接字 PF_PACKET 从内核缓冲区将数据包拷贝至用户空间缓冲区。

2.1.2 数据包捕获程序原理

使用 libpcap 进行抓包的基本步骤为：（1）打开网络设备（2）设置过滤规则（3）捕获数据包（4）关闭网络设备。libpcap 提供多种库函数实现上述功能，包括：

- 1) pcap_lookupdev()：函数用于查找网络设备，返回可被 pcap_open_live() 函数调用的网络设备名指针。
- 2) pcap_lookupnet()：函数获得指定网络设备的网络号和掩码。
- 3) pcap_open_live()：函数用于打开网络设备，并且返回用于捕获网络数据包的数据包捕获描述字。对于此网络设备的操作都要基于此网络设备描述字。
- 4) pcap_compile()：函数用于将用户制定的过滤策略编译到过滤程序中。

5) `pcap_setfilter()`: 函数用于设置过滤器。

6) `pcap_loop()`: 函数 `pcap_dispatch()` 函数用于捕获数据包, 捕获后还可以进行处理, 此外 `pcap_next()` 和 `pcap_next_ex()` 两个函数也可以用来捕获数据包。

7) `pcap_close()`: 函数用于关闭网络设备, 释放资源。

2.2 开源路由器 OpenWrt 介绍

2.2.1 OpenWrt 概述

OpenWrt 始于 2004 年, 第一个版本使用 LinkSys 源码。在 LinkSys 收费后, 改为由 Linux 内核继承。该系统是一个完全模块化且自动化程度高的嵌入式系统, 拥有强大的网络组件和可拓展性, 并且提供超过 100 个已经编译好的软件, 包括本研究所必需的数据包捕获函数库与对 `stdc++` 的支持。相比于其它路由器固件, OpenWrt 的优点是具有完全可写的文件系统, 用户可以自由的定义个性化应用程序, 且不需要在修改后重新编译整个系统。与传统 linux 系统相比, OpenWrt 在安装或编译软件时通过 feeds 机制提供了更简单的依赖库配置方案。传统 linux 系统需要在安装前手动安装所有的依赖库, 而在 OpenWrt 中只需要安装相应的模块化的 feed 软件包即可完成依赖库的配置。此外, OpenWrt 拥有基于 LUA 语言的图形化配置接口 LUCI。使用者可以直接使用浏览器通过路由器的 LAN 口地址访问 LUCI 界面进行网络配置。

2.2.2 OpenWrt 的网络适配器配置

OpenWrt 可以作为 client, 通过 WAN 口连接至其它 wifi, 再在 LAN 口创建自己的 wifi。因此, 在虚拟机上配置完毕的 OpenWrt 可以作为主机的路由器使用。下面介绍在 VMware 平台上配置 OpenWrt 虚拟机网络适配器的方式:

1) 配置虚拟网络。在 VMware 中编辑虚拟网络, 分别配置仅主机模式和 NAT 模式虚拟网卡的 IP 地址与子网掩码, 使其不发生冲突。



图 2-1： 虚拟网络配置

2) 为虚拟机添加两块网络适配器。其中第一块作为 OpenWrt 的 LAN 口网卡，通过仅主机模式与主机网络连接。第二块作为 OpenWrt 的 WAN 口网卡，通过 NAT 模式与互联网连接。

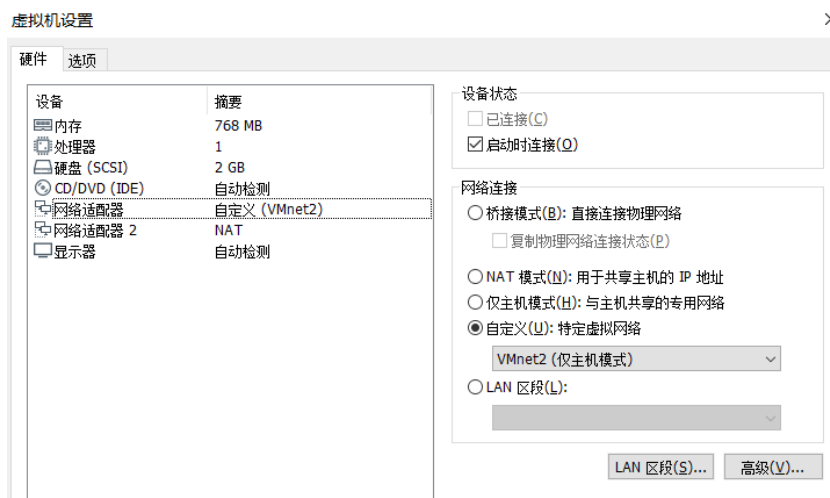


图 2-2-1： OpenWrt 虚拟机的网卡配置

配置完毕后开启 OpenWrt 虚拟机，打开位于 `/etc/config/network` 的网络配置文件并修改 LAN 口 IP 地址，使其处于上一步配置的仅主机模式虚拟网络中。

```
config interface 'loopback'
    option device 'lo'
    option proto 'static'
    option ipaddr '127.0.0.1'
    option netmask '255.0.0.0'

config globals 'globals'
    option ula_prefix 'fdcd:65b5:ddbc::/48'

config device
    option name 'br-lan'
    option type 'bridge'
    list ports 'eth0'

config interface 'lan'
    option device 'br-lan'
    option proto 'static'
    option ipaddr '192.168.126.99'
    option netmask '255.255.255.0'
    option ip6assign '60'

config interface 'wan'
    option device 'eth1'

- /etc/config/network 1/30 3%
```

图 2-2-2: OpenWrt 虚拟机的网络设置

3) 配置主机网络, 打开主机网络适配器编辑界面, 启用上一步配置的仅主机模式和 NAT 模式虚拟网卡, 并将它们的 IP 地址配置在上一步设定的网络中。此外, 还需将仅主机模式的网卡的默认网关设置为 OpenWrt 路由器的地址。



图 2-3-1: 主机网络适配器设置

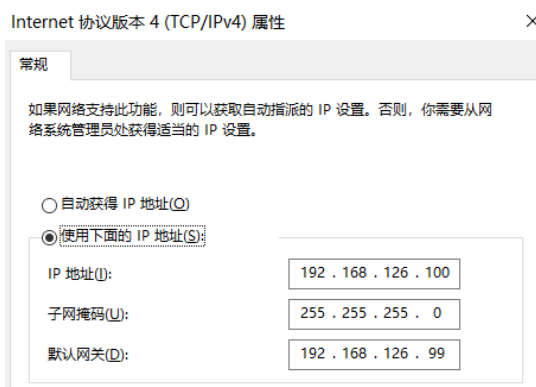


图 2-3-2: 主机仅主机模式网卡的 IPv4 协议设置

2.3 加密流量分类技术最新研究进展

2.3.1 视频流量预测方法

视频流量是目前 Internet 上占主导地位的流量，初步统计，网络上至少 82% 的流量是视频流量。现阶段对网络流量分类的研究主要集中于视频流量领域。Bronzino F, Schmitt P 以预测质量指标为目的，力图在复杂的网络场景——即视频流量与非视频流量交织，多种视频流量混合（例如，基于缓冲区的和基于吞吐量的速率自适应算法，固定大小的和可变大小的视频）的场景下，通过大量数据构成的训练集，实现一种基于流量分类的视频质量预测器，预测视频的启动延迟和分辨率。

2.3.2 QoE 预测方法

Mangla T, Halepovic E 等^[8]则以分析用户感知体验质量（QoE）为分析目标，提出一种名为 eMIMIC 的方法，通过采集无源网络中加密流量的包头，平均比特率和重缓冲率（re-buffering rate）作为训练集，以进行 QoE 度量与预测。并证明了该方法至少具有 89.6% 的准确性。

Mazhar M H, Shafiq Z^[4]设计了一套网络和传输层头部信息的综合特征。在传输层，考虑 TCP 标志位，重传次数等特征；在网络层，由于网络运行商无法观察到类 tcp 传输层的 QUIC 特性，因此进一步采用基于到达时间，包大小，字节计数和吞吐量的网络层特征。并在此基础上，提出了一种基于监督学习的具有较高实时性的 QoE 指标——如加密视频流的启动延迟，拒绝缓冲事件和实时视频质量——的推断方式。

2.4 本章小结

本章简要介绍本系统依赖的关键技术及其在本研究中的使用方式。包括包捕获函数库 libpcap 与开源路由器 OpenWrt。并介绍在加密流量分类方面当前的研究进展，以及相关技术。

第三章 系统需求分析

3.1 系统总体需求

针对任务书中描述的目标进行需求分析。轻量级流量特征器的应用场景如下：

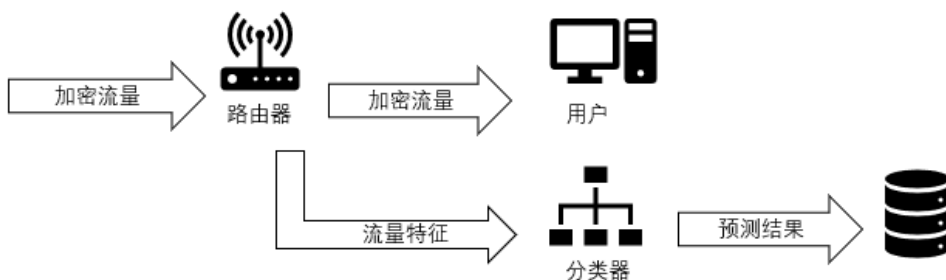


图 3-1：应用场景示意图

流量特征器运行于路由器中，对经过路由器的流量进行处理，实时监测数据率，首部信息等数据。而不对包有效载荷进行嗅探，也不改版包本身的内容。部署了加密流量特征器的路由器对网络应用用户仍是透明的。此外，流量特征器产生流量特征，交由分类器进行分类，并产生最终的预测结果。

根据需求方向，可将特征器的功能分为数据包抓取、包信息采集、流量特征生成三个主要部分。需完成从实时网络流量或 pcap、pcapng 文件中获取数据包、采集并存储数据包信息、计算并输出流量特征的设计要求。

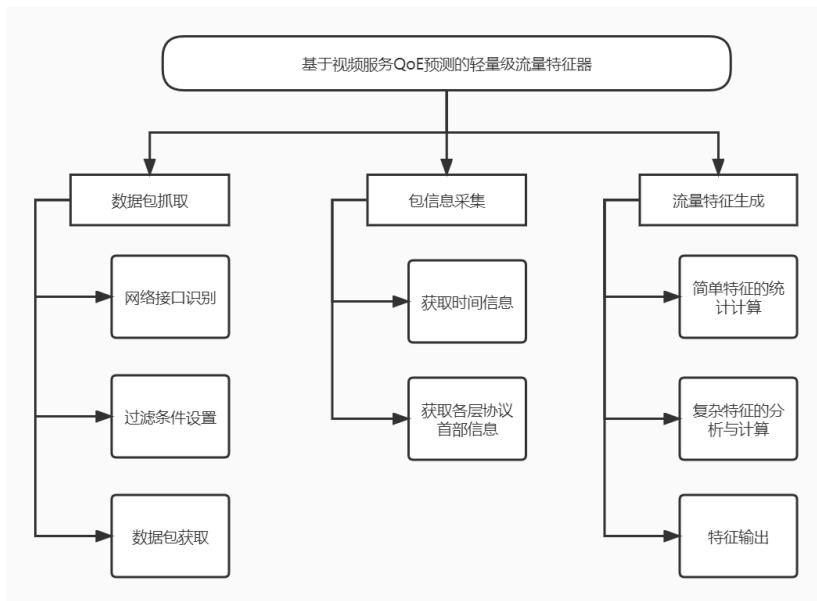


图 3-2：系统功能架构图

3.2 系统功能需求

3.2.1 数据包获取功能

进行流量特征提取，首先需要完成数据包的获取。基于任务书的要求，本研究需要完成静态和动态流量特征器的设计与实现。因此，需要设计从抓包结果文件；以及从经过路由器的实时流量中获取数据包的方法。

由于网络环境复杂，本研究在进行数据包获取前应进行预处理，通过设置过滤条件忽视部分无效、或于视频流量 QoE 预测无意义的数据包，避免结果受到干扰。此外，对于静态数据包获取，应提供文件选择功能。对于实时数据包获取，应提供网络接口选择功能。

3.2.2 包信息采集功能

包信息采集功能在包到达时获取其时间戳并存储。此外，应根据获取到的首部信息对数据包进行初步判断并分析其协议类型，以保证后续过程准确无误。首先应提取其数据链路层首部信息，根据首部协议类型字段识别上层协议。如果上层协议是 IPv4 协议，那么进行下一步首部获取。获取到 IPv4 首部内容后，根据其上层协议类型字段判断上层协议是否是 UDP 或 TCP 协议。如果不是，则将其丢弃，否则保存其首部字段以供特征提取，随后根据传输层协议类型进行传输层首部字段的保存。

3.2.3 流量控制功能

为保证对一条流量的特征实时，持续性地提取，需要单独维护每一条流量的数据，因此需要设计流量控制功能，创建并维护表示流量的类对象。在新的数据包到达时，能够识别出该数据包所属的流量及方向，并将其添加至所属流量中。在一条流量超时，应将其释放，避免内存空间的无限消耗。此外，流量控制层还应实现对流量自身的辅助控制，包括将流量反向等。

3.2.4 流量特征生成功能

本设计的核心需求是流量特征生成功能。此功能模块下包含三个功能。首先，简单特征指可以直接从数据包首部信息中获取的特征，如包长，协议类型；以及通过简单计算可以获得的，如平均比特率。其次，复杂特征指无法直接从首部信息获取，需要进行

逻辑判断或纵观整体流量信息的，如重传次数、段请求间隔。最后，在每个包到达时、或在每个段请求发出时，输出最新的流量特征。

3.2 非功能需求分析

性能需求包括：空间占用方面，流量特征器运行于 OpenWrt 路由器，该路由器内核内存空间为 16M，磁盘空间 120M。因此，需要充分优化代码参数传递过程。实时性要求方面，基于网络流量的特点，本系统具有较高的实时性要求，需要优化代码性能，提高运算和存储效率，避免数据积压或丢失。

可靠性需求包括：软件设计时应充分考虑系统可靠性，将故障频度控制在可接受的范围内。在实际使用中，保证正常工作时间占比达到 99%。

易用性需求包括：软件应具有简单的使用方式，保证输出信息可阅读。同时整体代码结构应简单，易于学习。

可移植性需求包括：软件源码应基于 linux 内核，减少不必要的动态库函数的使用，确保经交缠编译后，可以在不同的基于 linux 内核的嵌入式路由器系统中使用。

可维护性需求包括：软件应便于测试、便于根据需求变化进行适当更改。源码模块应保证高内聚低耦合。

3.3 本章小结

本章进行了需求分析。明确软件所需的各项功能，包括数据包抓取、包信息采集、流量特征生成这三个主要功能模块。并且明确了非功能需求，为后续系统设计奠定基础。

本研究设计的流量特征器用户的用例图如下：

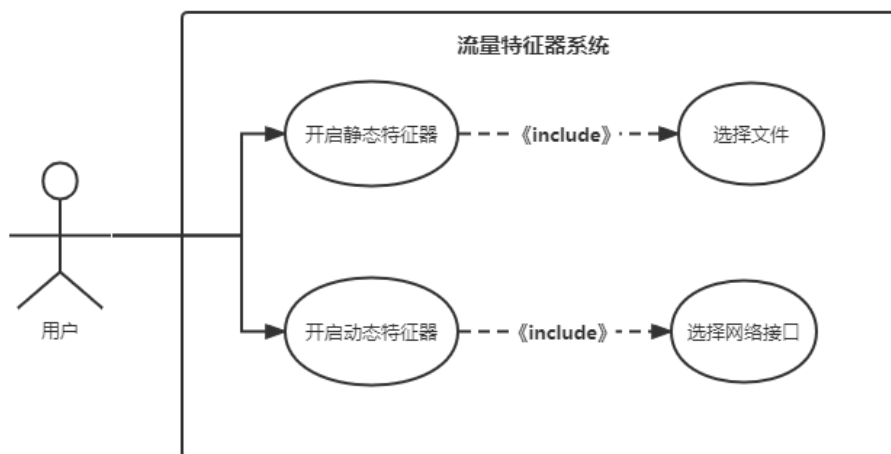


图 3-3：用户用例图

第四章 概要设计

4.1 系统结构设计

根据系统需求分析得到的系统功能需求，进行系统结构设计。



图 4-1：系统结构图

系统整体分为四层，分别是数据包获取层，信息处理层，流量管理层和特征计算层。下面将分别对各层进行接口设计及概要设计。

4.2 接口设计

1) 数据包获取层与信息处理层间接口：

信息处理层需要数据包获取层提供数据包内容，并由此完成首部信息和时间信息的处理。该传递机制由 libpcap 库函数 pcap_loop() 实现，将在详细设计部分具体描述。

2) 信息处理层与流量管理层间接口：

流量管理层依赖数据包首部信息进行数据包所属流量的判断。信息处理层通过为流量管理类的临时变量直接赋值完成首部信息和事件信息的传递。

3) 流量管理层与特征计算层间接口：

特征计算层同样依赖首部信息进行特征计算，除此之外还需要数据包所属流相关信息。特征计算层将每条流量绑定至一个 flow 类对象（详见附录 1），流量管理层负责创建 flow 类对象，并通过调用其类方法向其中添加数据包。

4.3 概要设计

4.3.1 数据包获取层

本层使用 libpcap 库函数完成网络接口的识别、过滤条件的设置，以及数据包的获取。

一般，路由器具有多个网络接口。以配置完毕的 OpenWrt 路由器为例，它具有 4 个网络接口，分别是：

- 1) br-lan：路由器的 LAN 接口。通过此接口路由器与主机连接并进行包转发。
- 2) eth0：路由器的 WAN 接口（ipv4），连接 Internet 或其它远端外围网络设备。
- 3) eth1：路由器的 WAN 接口（ipv6）。
- 4) lo：路由器的本地环回接口。

进行抓包，需要对 br-lan 网络接口进行监控。使用 libpcap 库中 pcap_lookupdev() 方法可自动获取合适的网络接口名称。并在后续过程中使用。设置过滤条件需要完成三步：构造过滤表达式，编译表达式，应用过滤器。过滤表达式是一种符合 tcpdump 过滤语法的表达式，编译过滤表达式，需要使用 pcap_compile() 函数。在编译的同时可以选择是否对过滤条件进行优化。最后，应用 pcap_setfilter() 方法部署过滤器。后续抓包工作则会按照过滤要求执行。数据包获取需要使用 pcap_loop() 方法。此方法从给定网络接口（或文件描述符）循环抓包，并交由指定 user 处理。

4.3.2 信息处理层

本层需完成数据包时间信息，以及头部信息的获取、处理与存储。

在抓取时，libpcap 会为每个包构造如下数据结构。

```
struct pcap_pkthdr
{
    struct timeval ts;    /* time stamp */
    bpf_u_int32 caplen;  /* length of portion present */
};
```

```

    bpf_u_int32 len;      /* length this packet (off wire) */
};

```

其中，ts 即为该数据包到达网络接口并被抓取时的时间戳。通过读取该数据结构即可获得数据包的时间信息。数据包的内容则会在抓取时以网络字节序全体传递给 pcap_loop() 方法的 user。本研究需要从数据链路层首部中获取协议类型字段，判断网络层协议是否为 IPv4；从 IPv4 首部获取源、目的 IP 地址，还需获取上层协议类型，判断传输层协议是否为 TCP 或 UDP；从 TCP 首部获取源、目的端口号，标志位信息，窗口长度等信息；从 UDP 首部获取源、目的端口号。

为获取以上首部信息，只需利用各层协议首部长度固定且各个字段数据对齐的特点（忽略 IPv4 首部附加字段，TCP 首部附加字段对提取信息无影响），按照各层协议首部结构定义数据结构，再使用移位的包内容对定义好的首部数据结构进行赋值，即可完成首部信息的获取。

4.3.3 流量管理层

网络流量最常用的唯一标识方式是采用五元组<源端口号,目的端口号,源 IP 地址,目的 IP 地址,应用层协议类型>。在本研究中，考虑到预测某种网络服务 QoE 往往需要获取该服务跨越多个协议的各种报文的特征，因此采用修改的五元组<源端口号,目的端口号,源 IP 地址,目的 IP 地址,流向>对网络流量进行唯一标识。本研究使用流关键字（flow_key）命名上述修改的五元组。使用流标记的优势是同一条链路上交互的不同协议信息将被划分在同一流量中，便于对各协议占比进行分析。进而推断网络状况。此外，对上行和下行流量作出区分，有利于在视频服务这一特殊领域借助网络层，传输层特征推断经过加密的应用层信息，这是传统加密流量特征提取无法实现的。本层还应实现流量超时和反向的判断。当一条数据包到达时，应根据其时间戳与其所属流量的最近时间戳进行对比。如果时间差超过阈值，则应判断它所属的流量超时，应将其释放并新建流量来存储当前包的信息。本层通过读取本研究设计的表示流量的类对象中的标志位来判断流量是否反向，若流量反向，则应将本来标记为正向的流量重新标记为反向，反之亦然。

本层借助信息处理层获得的包首部信息生成每个数据包的流关键字，并判断该包是否属于某个正在统计特征的流量。若是，则将其加入该流量；否则为它新建流量并存储该流标记。

4.3.4 特征计算层

本层完成流量特征的计算与输出。且根据所提取的特征进行简单 QoE 预测与简单流量分类。传统流量特征器对每个包的首部信息，以及各个时间点的流量特征进行持久化存储。在磁盘大小仅 120M 的 OpenWrt 路由器上，这样的特征器往往运行三十分钟便会崩溃。受限环境，本研究采用部分缓存、实时输出与延时计算的策略降低系统负载，达成轻量级的目标。临时部分缓存，指不对包的整体首部进行缓存。在特征计算的过程中，仅判别需要获取历史数据包首部信息。为了判别重传，仅需要获取历史数据包 TCP 首部的 ack 字段。因此，在流量计算的过程中仅需存储历史 TCP 首部 ack 信息，大大减少了存储空间占用。此外，本研究采用每当一个新包到达便进行一次特征输出的策略，为每个特征仅维持少数计算需要的数据。可以避免系统中大量缓存空间的消耗。延时计算指不在特征生成时进行复杂的统计量计算，转而计算那些可以在后续过程中计算出均值、方差、标准差等统计量而运算量小的数据。比如个数、线性和与平方和。这种方式可以加速特征计算，提高系统效率与实时性。

4.4 数据流程图

基于系统需求分析与概要设计，本系统的数据流程图如下：

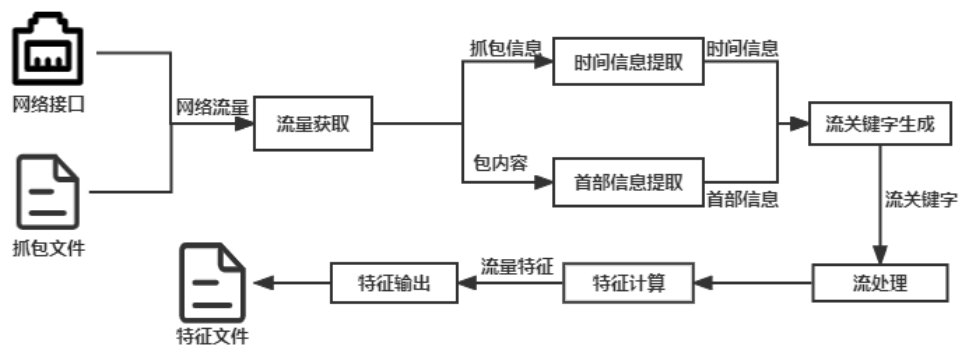


图 4-2：数据流程图

4.5 本章小结

本章进行系统概要设计，介绍系统层次结构，每层的功能并简要描述实现方式。最后得到系统的数据流图。

第五章 系统详细设计与实现

本章按照系统结构层次对每层中各个功能模块的具体逻辑和实现进行详细介绍，并详细阐述程序中的类及类方法定义

5.1 数据包获取层的设计与实现

5.1.1 libpcap 库函数的详细说明

实现数据包获取功能首先需进行 libpcap 动态库的配置^[9]。在 Ubuntu 系统中，执行命令：`sudo apt-get install libpcap-dev` 可以自动获取 libpcap 最新版本代码包并自动完成配置。本文第二章对 libpcap 库函数进行了简要介绍，下面详细说明本研究使用的各个库函数^[10]。

1)

表 5-1 libpcap 库函数介绍 1

<code>char * pcap_lookupdev(char * errbuf)</code>		
功能说明:		自动获取网络接口
返回值说明:		可用的网络接口字符串
参数说明:	errbuf	存放错误信息

本函数完成网络接口字符串的自动获取，若出错，则在 PCAP_ERRBUF_SIZE 长度的字符串中存放错误信息。

2)

表 5-2 libpcap 库函数介绍 2

<code>pcap_t * pcap_open_live(const char * device, int snaplen, int promisc, int to_ms, char * errbuf)</code>		
功能说明:		打开网络接口并设置抓包模式
返回值说明:		pcap_t 指针[11], 包含网络接口信息及抓包信息
参数说明:	device	网络接口字符串
	snaplen	对于每个数据包，抓取头部字节数量
	promisc	是否开启混杂模式，0 表示否，1 表示是
	to_ms	等待时间，为 0 表示一直等待到有数据包到达
	errbuf	存放错误信息

本函数实现打开 device 指定的网络接口，device 可以由 pcap_lookupdev 自动获取，或者通过硬编码写入函数。返回包含在 pcap_t 结构体中的接口描述符和接口信息。

同时可对抓包方式进行简单设置。其中 promisc 指定是否在网络端口开启混杂模式^[12]，使接口获取所有经过它的流量，无论该数据包的源或目的地址是否为它本身，是一种适合进行网络分析的网卡模式。

与本方法类似的，有函数 `pcap_t *pcap_open_offline(char *fname, char *ebuff)`，从离线文件，即.pcap 或.pcapng 文件中获取流量。由于是离线版本，故不涉及抓包长度、是否开启混杂模式，以及等待时间的问题。

3)

表 5-3 libpcap 库函数介绍 3

<code>int pcap_loop(pcap_t * p, int cnt, pcap_handler callback, u_char * user)</code>		
功能说明:		从网络接口循环抓包
返回值说明:		是否成功
参数说明:	p	打开网络接口函数获得的pcap_t类型指针
	cnt	需要抓取的包数。负数表示无限循环
	callback	回调函数指针
	user	传递给callback的参数

在打开的网络接口上依次进行抓包，并交由 callback 函数指针指向的函数处理。同时将 user 作为 callback 的参数传递。callback 函数的具体定义将在下一部分进行。

表 5-4 libpcap 库函数介绍 4

<code>int pcap_compile(pcap_t *p, struct bpf_program *fp, char* str, int optimize, bpf_u_int32 netmask)</code>		
功能说明:		编译过滤语句
返回值说明:		是否成功
参数说明:	p	打开网络接口函数获得的pcap_t类型指针
	fp	传出参数，存放编译后的bpf语句[13]
	str	过滤表达式
	optimize	是否进行优化
	netmask	网络掩码，通常设置为0

<code>int pcap_setfilter(pcap_t * p, struct bpf_program * fp)</code>		
功能说明:		设置过滤器
返回值说明:		是否成功
参数说明:	p	打开网络接口函数获得的pcap_t类型指针
	fp	编译后的bpf语句

以上两函数完成过滤语句的编译与配置。其中，bpf 语句指满足 BSD Packet Filter 结构的语句。BSD Packet Filter 结构是 BSD，linux 等操作系统内核提供的数据包过滤方法。libpcap 同样利用 BPF 过滤数据包。若希望仅保留 TCP 和 UDP 数据包，则应编写 BPF 语句 “tcp and udp”。

BPF 语句编写完成后，使用 pcap_compile 函数在内核中完成编译，随后使用 pcap_setfilter 函数即可完成部署。

5)

表 5-5 libpcap 库函数介绍 5

void pcap_close(pcap_t * p)		
功能说明:		关闭网络接口并释放资源
返回值说明:		无
参数说明:	p	待关闭的网络接口

关闭网络接口函数。抓包完毕后必须关闭打开的网络接口，否则将出现内存泄漏。

5.1.2 数据包获取模块设计

数据包获取层数据流程图如下：

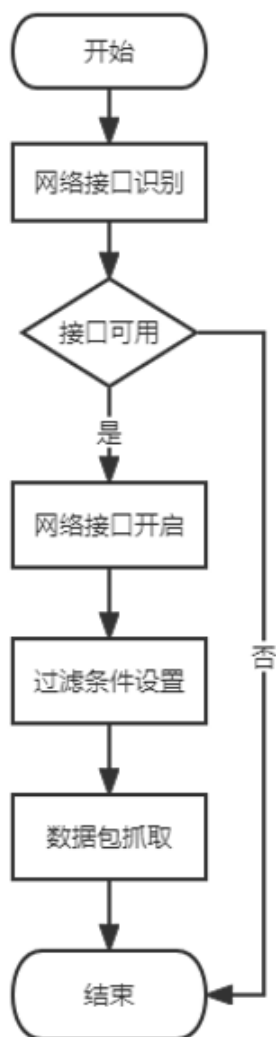


图 5-1 数据包获取层数据流程图

该层依次完成获取网络接口，打开网络接口，设置过滤器，抓包与关闭网络接口。在打开网络接口步骤设置获取数据包前 65536 字节的内容。在设置过滤器步骤设置过滤条件为进获取 tcp 与 udp 包。在抓包步骤设置循环抓包，直到发生错误或程序中止。

5.2 信息处理层的设计与实现

信息处理层在本研究中有重要作用。所有与特征计算相关的信息均在此层进行提取与处理。本层通过上一部分中介绍的 pcap_loop 函数中 callback 函数指针指向的函数

来获取数据包捕获层产生的抓包结果。定义数个首部信息结构体存储数据包各层协议首部信息，并以此为流量管理层 flow_process 类对象赋值，将获取到的信息传递给后续功能模块：

1) 以太网首部结构体

```
struct ethernet_header
{
    u_int8_t ether_dhost[6]; /*目的以太网地址*/
    u_int8_t ether_shost[6]; /*源以太网地址*/
    u_int16_t ether_type; /*以太网类型*/
};
```

2) IPv4 首部结构体

```
struct ip_header
{
#ifdef WORKS_BIGENDIAN
    u_int8_t ip_version : 4,
    ip_header_length : 4; /*首部长度*/
#else
    u_int8_t ip_header_length : 4,
    ip_version : 4;
#endif

    u_int8_t ip_tos; /*服务类型*/
    u_int16_t ip_length; /*总长度*/
    u_int16_t ip_id; /*标识*/
    u_int16_t ip_off; /*片偏移*/
    u_int8_t ip_ttl; /*生存时间*/
    u_int8_t ip_protocol; /*协议类型*/
    u_int16_t ip_checksum; /*首部检验和*/
    struct in_addr ip_source_address; /*源IP*/
    struct in_addr ip_destination_address; /*目的IP*/
};
```

3) UDP 首部结构体

```
struct udp_header
{
    u_int16_t udp_source_port; /*源端口号*/
    u_int16_t udp_destination_port; /*目的端口号*/
};
```

```

    u_int16_t udp_length;  /*长度*/
    u_int16_t udp_checksum; /*校验和*/
};

```

4) TCP 首部结构体

```

struct tcp_header
{
    u_int16_t tcp_source_port;          /*源端口号*/

    u_int16_t tcp_destination_port;    /*目的端口号*/

    u_int32_t tcp_acknowledgement;     /*序号*/

    u_int32_t tcp_ack;                  /*确认号字段*/
#ifdef WORDS_BIGENDIAN
    u_int8_t tcp_offset : 4,
    tcp_reserved : 4;
#else
    u_int8_t tcp_reserved : 4,
    tcp_offset : 4;
#endif
    u_int8_t tcp_flags;
    u_int16_t tcp_windows;              /*窗口字段*/
    u_int16_t tcp_checksum;             /*检验和*/
    u_int16_t tcp_urgent_pointer;       /*紧急指针字段*/
};

```

上述结构体中成员变量生命严格采取 u_int16_t 或 u_int32_t 形式，目的是保证数据对其方式与协议首部一致。这样，使用如下赋值语句，以报文内容 packet 对各个首部结构体赋值，即可保证首部结构体中各个字段的值与协议首部相同：

1. ethernet_protocol = (struct ethernet_header*)packet;
2. ip_protocol = (struct ip_header*)(packet + 14);
3. tcp_protocol = (struct tcp_header*)(packet + 14 + 20);

在 5.2 节中介绍了 pcap_loop 函数，从打开的网络接口或离线文件中循环读取数据包并交由 callback 函数处理。且 pcap_loop 函数的最后一个参数 `uchar* user` 将作为 callback 函数指针指向的函数的第一个参数传递给 callback。下面详细介绍 callback 指向的函数应满足的格式与具体实现。在本研究中，callback 函数名为 `getPacket`：

表 5-6 callback 函数介绍

<pre>void getPacket(u_char * user, const struct pcap_pkthdr * pkthdr, const u_char * packet)</pre>		
功能说明：		对接受到的数据包进行处理
返回值说明：		返回网络接口字符串
参数说明：	<code>user</code>	pcap_loop 传递的参数
	<code>pkthdr</code>	存储数据包的时间信息和长度信息的结构体
	<code>packet</code>	数据包数据

信息处理层需要完成时间信息与包首部信息的提取。其中时间信息只需读取 `pkthdr` 结构体即可获取。而通过上述结构体的赋值，也可完成首部信息的获取。此外，本研究通过 `user` 参数实现了外围包计数功能，定义初值为 0 整型变量 `id` 作为 `user` 参数传入 `pcap_loop`，进而传递给 callback 函数，并在 callback 中进行累加。由此，可在每个新包到达时输出其序号。本层的数据流程图如下：

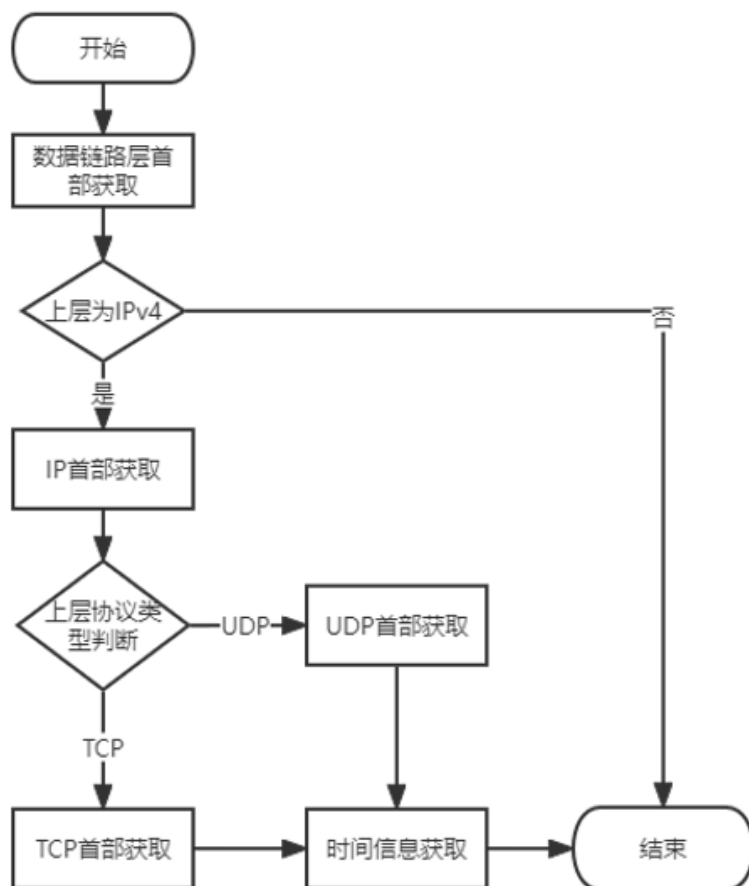


图 5-2 信息处理层数据流程图

5.3 流量管理层的设计与实现

流量管理层记录已有流量的流标记，并暂存新到达包的首部信息，用以判断该包是否属于已有流量。此功能通过 flow_process 类实现。下面详细介绍该类及类方法的设计与实现。首先引入流关键字结构体与首部信息结构体：

```
struct flow_key
{
    struct in_addr ip_source_address; /*源IP*/
    struct in_addr ip_destination_address; /*目的IP*/
    u_int16_t tcp_source_port; /*源端口号*/
    u_int16_t tcp_destination_port; /*目的端口号*/
    FlowDirection flow_direction; /*枚举类型，包含前向和后向*/
    std::string to_string() const;
    bool operator < (const struct flow_key& e) const;
};
```

流关键字由源、目的 IP 地址，源、目的端口号和流方向构成，且包含方法 to_string 及重载运算符<。to_string()的功能是将流关键字五元组依此转化为字符串并拼接，而<用于判断两个经过 to_string 的流关键字是否是相同的字符串。to_string 和<结合使用可以实现判断一个包是否属于某条已有流量的功能。

```
struct simple_packet_info
{
    struct flow_key flow_key;
    time_type ts; /* time stamp */
    bpf_u_int32 packet_len; /* length of this packet (off wire) */
    FlowDirection flow_direction;
    int protocol;
    struct ip_header* ip_protocol;
    struct tcp_header* tcp_protocol;
    struct udp_header* udp_protocol;
};
```

包首部信息类包含数据包的流关键字，时间戳，包长以及流向。通过整型变量 protocol 记录传输层协议类型号。若为 6，则传输层为 udp 协议，udp_protocol 存储 udp 首部，而此时 tcp_protocol 为 NULL；若为 17，则传输层为 tcp 协议，tcp_protocol 存储 tcp 首部，而此时 udp_protocol 为 NULL。

下面介绍 flow_process 类：

```
class flow_process
{
public:
    std::map <struct flow_key, flow>flows;

    struct simple_packet_info simple_packet_info;

    flow_process();

    void on_packet_received();

};
```

表 5-6 flow_process 类介绍

class flow_process		
成员变量：	flows	流关键字到流的映射
	simple_packet_info	暂存的新到达包的首部信息
成员函数：	on_packet_received()	对新到达包的处理

flow_process 类的成员函数 on_packet_received() 完成对新到达的包的一些列处理，包括所属流量判断，流向判断，新流量的生成与记录或者向现有流量中添加数包。用数据流程图展示其实现逻辑：

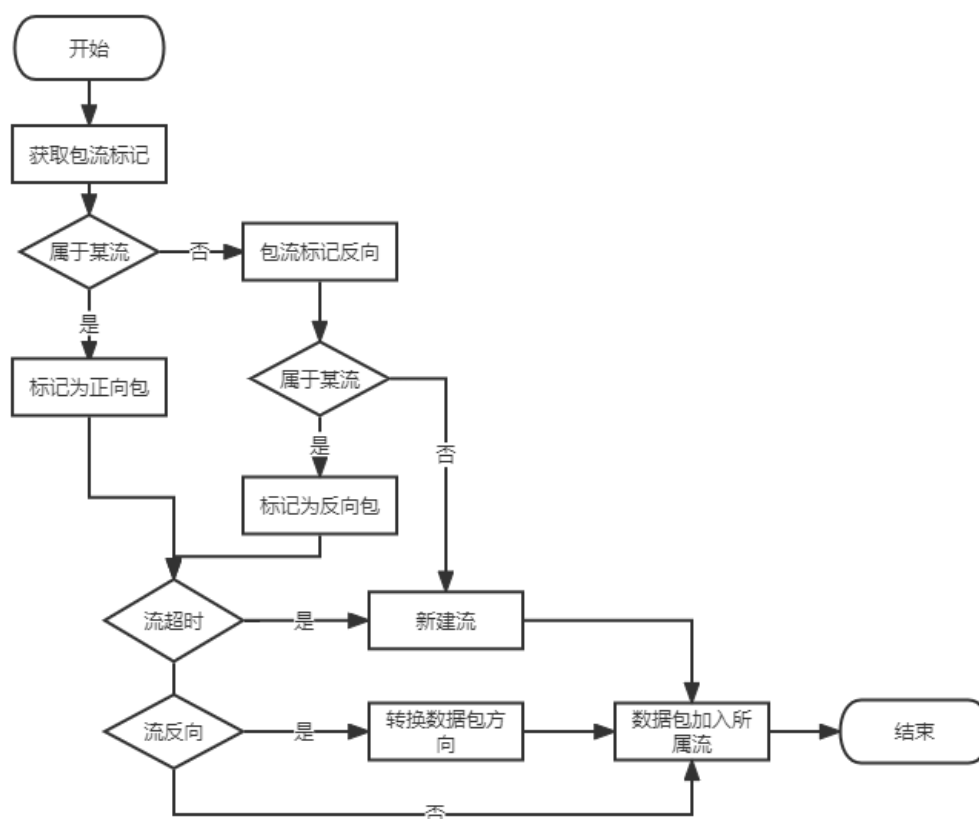


图 5-3 流量管理层数据流程图

- 1) 通过流关键字判断数据包是否是某个已有流量的正向数据包。若否，转到 2)。若是，转到 6)
- 2) 反转数据包流关键字中的源、目的 IP 地址和源、目的端口号。通过反转的流关键字判断数据包是否是某个已有流量的反向数据包，若否，转到 3)。若是，转到 5)
- 3) 新数据包不属于已有流量，恢复其流关键字。转到 4)
- 4) 以当前数据包的流关键字创建新的流。将新数据包方向设置为正向。转到 9)
- 5) 新数据包是某个已有流量的反向数据包，恢复其流关键字。将新数据包方向设置为反向。转到 7)
- 6) 新数据包是某个已有流量的正向数据包。将新数据包方向设置为正向。转到 7)
- 7) 判断数据包所属流是否超时，若超时，将该流释放，转到 4) 否则转到 8)
- 8) 判断数据包所属流是否反向，若反向，重新标记数据包方向，将其反转。转到 9)
- 9) 将数据包加入其所属流，结束。

上述逻辑实现了流量管理层的主要功能，完成新数据包的处理。

5.4 特征计算层的设计与实现

上一节介绍了流量管理层功能，却没有对流量进行明确定义。下面将详细介绍本研究中用于表示流量的类 flow，以及它用于特征计算的成员变量和成员函数。在此之前，需首先介绍本研究采取的特征计算和存储方法。

5.4.1 特征存储设计

对于每条流量特征，传统流量特征器往往会选择同时计算其全部统计特征。这会导致较大的运算量与内存占用。因此本研究采用增量特征的方式减少运算与内存占用，并附加衰减机制，使流量特征的提取更关注当前网络流量状态。增量特征的设计思路为：

假设 $s = \{x_1, x_2, \dots\}$ 是观察到的某特征的序列。通过维护三元组 $IS = (N, LS, SS)$ 可以逐步更新 S 的均值，方差和标准差。设 x_i 为新到达包的某个流量数据，按照 $IS \leftarrow (N + 1, LS + x_i, SS + x_i^2)$ 维护 IS ，之后可算出：

$$\text{均值 } \mu = \frac{LS}{N},$$

$$\text{方差 } \sigma^2 = \left| \frac{SS}{N} - \left(\frac{LS}{N} \right)^2 \right|,$$

$$\text{标准差 } \sigma = \sqrt{\left| \frac{SS}{N} - \left(\frac{LS}{N} \right)^2 \right|}.$$

为在此过程中附加衰减，引入衰减因子 $d = d(\lambda, t) = 2^{-\lambda t}$ 。其中， λ 为衰减参数，预设值为 1。 t 为上一个包到达至本包到达的时间。修改上述维护 IS 的过程为如下三步：

$$\begin{aligned} \gamma &= d(\lambda, t_{cur} - t_{last}) \\ IS &= (\gamma d, \gamma LS, \gamma SS, t_{cur}) \\ IS &= (d + 1, LS + x_i, SS + x_i^2, t_{cur}) \end{aligned}$$

之后可计算出：

$$\text{均值 } \mu = \frac{LS}{d}$$

$$\text{方差 } \sigma^2 = \left| \frac{SS}{d} - \left(\frac{LS}{d} \right)^2 \right|$$

5.4.2 特征计算设计

特征计算基于本层定义的表示单个流量的 flow 类实现，其中每个流由流关键字结构体唯一标识，该结构体中定义了流关键字的比较方法。

表 5-7 flow 类介绍

class flow（省略特征计算相关变量）		
成员变量：	flow_key	流关键字
成员函数：	add_packet()	由flow_process类方法调用，接收新包并提取特征
	decay()	计算衰减因子
	out_put_packet_c()	计算并输出特征
	terminate()	结束当前流，输出综合特征并回收空间
	to_reverse()	将当前流反向，交换其正、反向特征
	judge_vedio()	判断当前流量是否为视频流量、是否反向

本层的数据流程图如下：

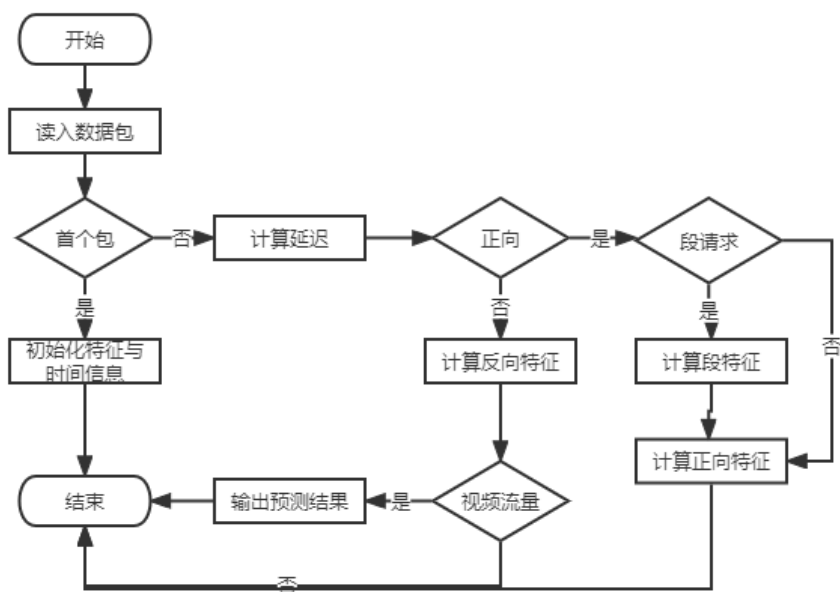


图 5-4 特征计算层数据流程图

当上节所述 flow_process 类对象接收新包，并为它分配所属流量后，会调用该 flow 的 add_packet() 方法，将该包加入它所属的流。随后，flow 将调用 decay() 方法计算衰减因子并进一步调用 out_put_packet_c() 方法计算特征。最后完成时间信息的更新。当当前流量特征的统计数据较为稳定时，out_put_packet_c() 函数将开启流量类型的判断，调用 judge_vedio() 方法，结合视频流量的特点，根据当前流量正、反向流量平均包长的特点判断流量是否为视频流量，以及是否为反向的视频流量。其具体逻辑为：若正向平均包长小于 100，而反向平均包长大于 1000，则判断为视频流量。若反向平均包长小于 100，而正向平均包长大于 1000，则判断为反向的视频流量。流量出现反向的原因是

流量控制层将该流量的第一个数据包方向标记为正向。而实际的正向应至上行流量。因此，需要对反向的流量及逆行方向转换，调用 `to_reverse()` 方法交换其正、反向流量特征并更改流关键字。

下面将详细介绍流量相关的变量及流量特征的提取，计算，输出方式。

本研究对同一流量中正向与反向流量分别进行特征提取，以正向流量为例介绍其中相同的部分（反向流量特征以 `bwd` 开头）：

```
time_type fwd_latest_timestamp; /*正向上一包到达时间*/

std::vector<packet_count_type> fwd_last_TCPseq; /*记录历史 seq*/

time_type      fwd_interarrival_time_n,      fwd_interarrival_time_ls,
fwd_interarrival_time_ss;

/*正向到达间隔的个数，线性和与平方和（带衰减）*/

double      fwd_packet_size_n,      fwd_packet_size_ls,      raw_fwd_packet_size_ls,
fwd_packet_size_ss;

/*正向包长的个数，线性和与平方和（带衰减）*/

double fwd_window_size_n, fwd_window_size_ls, fwd_window_size_ss;

/*正向窗口大小的个数，线性和与平方和（带衰减）*/

double fwd_flow_bps, fwd_flow_pps; /*平均比特率，平均每秒包数*/

time_type fwd_flow_duration; /*正向流量持续时间*/

time_type fwd_start_timestamp; /*正向第一个包到达时间*/
```

参考上述代码，本研究对包到达间隔，包长，窗口大小进行增量特征的计算。以到达间隔为例，其计算方式如下：

```
this->fwd_interarrival_time_n = decay * this->fwd_interarrival_time_n + 1;

this->fwd_interarrival_time_ls = (pkt.ts - this->fwd_latest_timestamp) +
this->fwd_interarrival_time_ls * decay;

this->fwd_interarrival_time_ss = (pkt.ts - this->fwd_latest_timestamp) *
(pkt.ts - this->fwd_latest_timestamp) + this->fwd_interarrival_time_ss * decay;
```

此外，对平均比特率，平均每秒包数以及持续时间进行简单统计，其方法如下：

```
double fwd_duration_temp = pkt.ts - this->fwd_start_timestamp;

if (fwd_duration_temp==0){

this->fwd_flow_bps = 0;

this->fwd_flow_pps = 0;

this->fwd_flow_duration=0;
```

```

    }

    else{

        this->fwd_flow_bps =
        (pkt.packet_len/fwd_duration_temp)+(this->fwd_flow_duration/fwd_duration_temp)*this
        ->fwd_flow_bps;

        this->fwd_flow_duration = fwd_duration_temp;

        this->fwd_flow_pps = this->fwd_pkt_count / this->fwd_flow_duration;

    }

```

在比特率计算层面，由于长时间尺度下记录所有数据包长度的总和会导致任何一种数据类型溢出，因此借助加权平均的思想对其进行间接计算，随后通过总包数与持续时间计算每秒包数。

上述部分介绍了简单流量特征的计算与提取。接着介绍复杂特征：重传次数及段相关特征的的计算。

对于重传次数，本研究将正向重传与反向重传合并计算，以便推断整个链路状态。重传的判断依据为同一方向 TCP 首部 seq 字段的重复。在 flow 类中，定义了

```
std::vector<packet_count_type> fwd_last_TCPseq; /*记录历史 seq*/
```

以记录正向的历史 seq 信息，以及它以 bwd 开头的相同定义 vector。

当新的 tcp 包到达时，检查它的 ack 字段是否与 last_TCPseq 中记录的 seq 相同。若相同，说明发生了重传。将 flow 类中定义的重传次数成员 packet_count_type retrans 累加。

对于段相关特征，其定义如下：

```

packet_count_type segment_count, segment_size_temp; /*段总数，段长临时变量*/

double segment_size_n, segment_size_ls, segment_size_ss;

/*段长数据的个数，线性和与平方和（带衰减）*/

time_type segment_req_interval_n, segment_req_interval_ls,
segment_req_interval_ss;

/*段请求间隔数据的个数，线性和与平方和（带衰减）*/

time_type last_req_time; /*上一个段请求到达时间*/

time_type
segment_duration_temp, segment_duration_n, segment_duration_ls, segment_duration_s
s;

/*段持续时间的个数，线性和与平方和（带衰减）*/

```

共包含三个段特征，即段长，段请求间隔与段持续时间，以及辅助计算这些数据的变量。

段数据属于应用层特征，无法直接从加密流量中提取。因此需要借助视频服务的特点，从网络层和传输层首部信息中推断其特点。视频服务的特点是，一旦服务开始，上行流量（正向流量）将只剩下段请求与 ACK 两种报文。在 TCP 协议中，ACK 报文的有效载荷小于等于 1。而在 QUIC 协议中，上行数据包中长度大于 150 字节的是段请求报文。由此便可以在正向流量中通过包长识别出段请求报文，进而计算段特征。具体逻辑如下：

- 1) 新包到达，判断其方向，若为正向，转到 2)；若为反向，转到 4)
- 2) 结合协议判断是否为段请求，若是，转到 3)；否则转到 5)
- 3) 计算段持续时间，更新上一次请求时间变量 `last_req_time`，计算段增量特征并输出，清空段长和段持续时间临时变量。转到 5)
- 4) 更新段长和段持续时间临时变量。转到 5)
- 5) 继续。

基于上述方法，本研究实现了对多方面，多协议层次的流量特征的轻量级提取。

5.5 本章小结

本章详细介绍系统各层的设计与实现方式。

其中数据包获取层通过调用 libpcap 库函数实现。信息处理层根据各层协议首部结构定义了数个首部信息结构体，完成对首部信息的提取。流量管理层通过流关键字结构体对流量进行方向判断与归属管理。最后，特征提取层基于增量特征与衰减的思路完成对多种流量特征的提取与输出。

本研究的整体结构图如下：

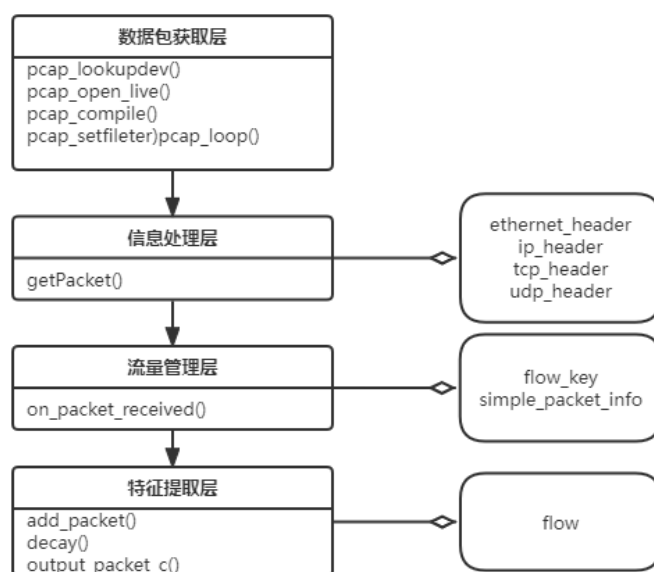


图 5-5：整体结构图

第六章 软件部署与测试

本研究在 Ubuntu 系统下编码，在本机编译产生的可执行文件无法直接在 OpenWrt 路由器上使用。因此需配置交叉编译环境并进行交叉编译。

6.1 软件部署

本研究的编码工作在 Ubuntu 系统中进行，系统指令集架构、数据存储方式均与目标及 OpenWrt 系统不同。因此需要进行交叉编译。交叉编译主要分为 4 步：配置宿主机交叉编译环境，配置虚拟机依赖库，宿主机交叉编译动态库，源码交叉编译与移植。

6.1.1 宿主机交叉编译环境配置

在 OpenWrt 官方下载站点可以找到与某版本 OpenWrt 版本系统对应的 SDK 工具。下载解压可得交叉编译器、交叉编译工具链及部分依赖包。本研究使用 x86_64 架构，大端存储方式的 OpenWrt 系统，对应的交叉编译器为 x86_64-openwrt-linux-g++/gcc。将交叉编译器所在文件夹加入环境变量 PATH 中，即可作为类似 g++/gcc 的命令使用，进行源码的交叉编译。

6.1.2 虚拟机动态库配置

动态库是一类函数库，在目标文件链接阶段指明，随后与目标文件保持独立。目标文件执行时仍需要该动态库支持。OpenWrt 支持百种以上预编译好的软件包，包括本研究的代码所需的 libpcap 和 stdc++6 库。OpenWrt 提供 opkg 命令，用于从官方网站获取软件包并安装。开机后首先执行 opkg update 命令更新下载源，随后分别执行 opkg install libpcap 和 opkg install stdc++6 安装 libpcap 和 stdc++6 动态库。打开 usr/lib 查看下载的动态库版本号，随后需要在宿主机中配置相同版本动态库。由图 6-1 可以看出，该系统支持的 OpenWrt 版本为 1.9.1，支持的 stdc++6 版本为 6.0.25。

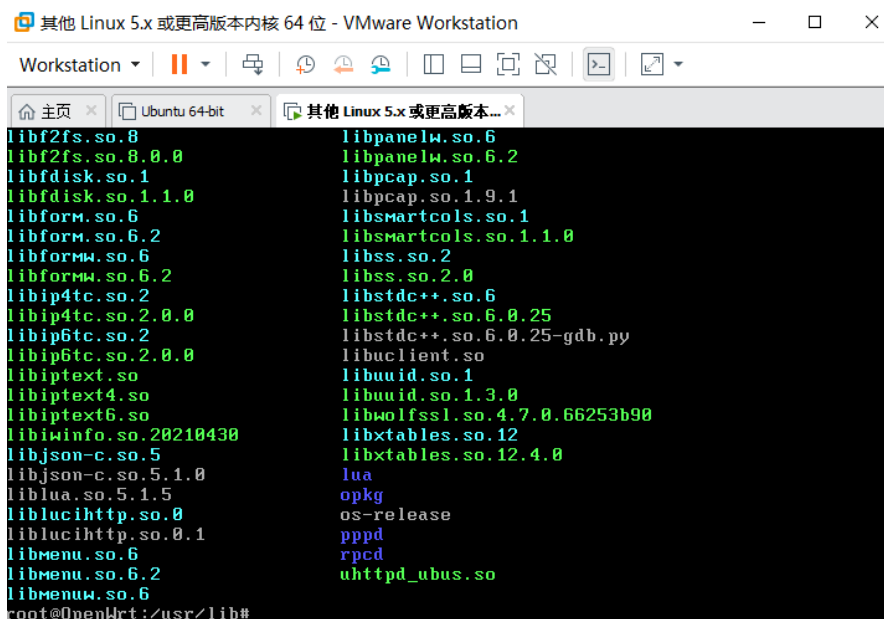


图 6-1: OpenWrt 中的动态库

6.1.3 宿主动态库的交叉编译

在宿主机中打开所下载的 OpenWrtSDK 中依赖库文件夹，确认 SDK 携带的 libc++6 库与目标机一致。随后进行 libpcap 库的交叉编译^[14]。

从 tcpdump 官网中找到 1.9.1 版本的 libpcap 库源码，在 Ubuntu 中解压后进入文件夹。首先编写 shell 程序 env.sh 配置环境变量：

```
1 Workdir=~/.openlibpcap
2 export CROSS=i486-openwrt-linux-
3 export CC=${CROSS}gcc
4 export STRIP=${CROSS}strip
5 export AR=${CROSS}ar
6 export RANLIB=${CROSS}ranlib
7 export OBJCOPY=${CROSS}objcopy
```

图 6-2: 配置 libpcap 库交叉编译环境变量的 shell 程序

执行 source ./env.sh 后完成环境变量的配置。此时使用 echo \${变量名} 即可确认环境变量已改变。接下来依此执行以下命令：

./configure --prefix=/home/libpcapso --host=x86_64-openwrt-linux，其中 --prefix 指定生成动态库的文件夹。若不指定，默认在 /usr/lib 中生成动态库。由于本研究中宿主机已配置好 libpcap 用以调试，因此指定新文件夹存放交叉编译产生的动态库。--host 指明此次编译为交叉编译，目标机为 x86_64-openwrt，而宿主机为 linux。

make && make install，进行编译与安装。

命令执行成功后，检查 libpcapso 文件夹，得到 libpcap.so.1.9.1 动态库文件。将其拷贝至交叉编译工具链的依赖库文件夹中，即可在编译时使用 -lpcap 参数进行指明。

6.1.4 源码交叉编译与移植

使用 x86_64-openwrt-linux-g++ 交叉编译器对源码进行交叉编译。源码文件夹如下：

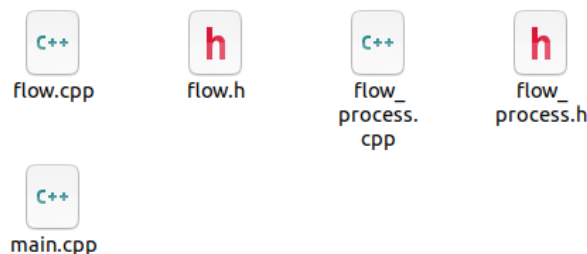


图 6-3 流量特征器源码文件夹

按序执行以下命令完成源码交叉编译，并产生名为 fn 的可执行文件：

```
x86_64-openwrt-linux-g++ -c flow_process.cpp -o flow_process.o -lpcap
```

```
x86_64-openwrt-linux-g++ -c flow.cpp -o flow.o -lpcap
```

```
x86_64-openwrt-linux-g++ -c main.cpp -o main.o -lpcap
```

```
x86_64-openwrt-linux-g++ flow_process.o flow.o main.o -o fn -lpcap
```

交叉编译成功后，通过 SCP 协议将可执行文件 fn 上传至 OpenWrt 中，赋予 fn 可执行权限，即可进行试运行。

6.1.5 错误示例

在配置交叉编译环境并进行编译的过程中，发生各种错误是不可避免的，下面描述部分典型错误。

1) 可执行文件格式错误：

```
root@OpenWrt:/error# ./f*
./fn_stdcpp_half: line 1: ELF4=b: not found
./fn_stdcpp_half: line 2: syntax error: unexpected "("
```

图 6-4：典型报错信息 1

图中”syntax error: unexpected ‘(‘”是可执行文件错误的典型报错信息。出现该报错是应检查所配置的交叉编译环境与目标机是否一致，即是否存在架构区别，数据的大小端存放方式是否一致等。

2) 目标机动态库版本不匹配：

```
Error loading shared library libpcap.so.1: Exec format error (needed by ./fn_live)
Error relocating ./fn_live: pcap_close: symbol not found
Error relocating ./fn_live: pcap_open_live: symbol not found
Error relocating ./fn_live: pcap_compile: symbol not found
Error relocating ./fn_live: pcap_setfilter: symbol not found
Error relocating ./fn_live: pcap_loop: symbol not found
```

图 6-5-1：经典报错信息 2

```

Error loading shared library libstdc++.so.6: Exec format error (needed by ./fn_1
live)
Error relocating ./fn_live: _ZSt20__throw_length_errorPKc: symbol not found
Error relocating ./fn_live: _ZdlPv: symbol not found
Error relocating ./fn_live: _ZdlPvm: symbol not found
Error relocating ./fn_live: __cxa_rethrow: symbol not found
Error relocating ./fn_live: _ZNKSt4sizeEv: symbol not found
Error relocating ./fn_live: _ZNSt4_Rp10_M_refdataEv: symbol not found
Error relocating ./fn_live: _ZSt18_Rb_tree_decrementPSt18_Rb_tree_node_base: sym
bol not found
Error relocating ./fn_live: _ZNSt4_Rp9_S_createEMMRKSaIcE: symbol not found
Error relocating ./fn_live: _ZNKSt7compareERRKS: symbol not found
Error relocating ./fn_live: _ZNSt13_S_copy_charsEPcS_S_: symbol not found
Error relocating ./fn_live: _ZNSt5D1Ev: symbol not found
Error relocating ./fn_live: _ZNSt6insertEMRRKS: symbol not found
Error relocating ./fn_live: _ZSt29_Rb_tree_insert_and_rebalancebPSt18_Rb_tree_no
de_baseS0_RS_: symbol not found
Error relocating ./fn_live: _ZNSt1cED1Ev: symbol not found
Error relocating ./fn_live: _ZSt19__throw_logic_errorPKc: symbol not found
Error relocating ./fn_live: _ZNKSt8capacityEv: symbol not found
Error relocating ./fn_live: __cxa_end_catch: symbol not found

```

图 6-5-2: 典型报错信息 3

图中报错信息表示无法引入 libpcap. so. 1 动态库或无法引入 stdc++6 动态库。但此时检查/usr/lib 文件夹，发现相应动态库已经存在。这是由于在目标机中未使用与交叉编译环境中相同的动态库版本。例如上述错误分别发生于宿主机使用 libpcap1.10.1, stdc++6.0.29; 而在目标机使用 libpcap1.9.1, stdc++6.0.25 导致的。可以发现，即使动态库大版本号一致，也不能保证程序的正常运行。

6.2 实验环境选择

本研究使用的各项技术框架的版本选择基于 OpenWrt 路由器架构的支持。为保证系统正常运行，应当保证和本研究使用相同的版本：

```

OpenWrt-21.02.0-x86-64_gcc-8.4.0_musl
libpcap. so. 1.9.1
stdc++. so. 6.0.25

```

其它开发环境列举如下：

```

Ubuntu22.04 LTS
vsCode 1.64
WinSCP 5.19.6

```

本研究在 linux 系统下进行开发与调试。进行 linux 下的调试运行时，动态库版本可以选择最新发行的稳定版。但在 OpenWrt 系统中运行时，应严格保证选取的动态库版本与 OpenWrt 支持的一致。否则会出现复杂的运行时错误。

6.3 软件测试

基于任务书的目标，本研究分别基于 `pcap_open_live()` 和 `pcap_open_offline()` 方法编写了动态与静态流量特征的数据包获取模块，并经过交叉编译生成 `fn_live` 和 `fn_offline` 可执行文件。

6.3.1 离线流量特征器的测试

为测试静态流量特征器，需要获取一段网络视频服务的抓包数据。由于特征器运行于路由器，无法保证单一网络环境。所以在生成测试文件时不应关闭其它网络服务，而应在其它网络服务正常运行的情况下打开视频服务进行抓包。

本测试使用 wireshark 抓包软件，主机通过 WLAN 接口连接互联网。在 wireshark 中选择网卡 WLAN 进行抓包。期间打开 `www.bilibili.com` 观看一则长 5 分钟的视频。视频结束后结束抓包。

10.128.205.212	120.92.84.136	TLSv1.2 Application Data
120.92.84.136	10.128.205.212	TLSv1.2 Application Data
10.128.205.212	120.92.84.136	TCP 49294 → 443 [ACK] Seq=32 Ack=28 Win=511 Len=0
10.128.205.212	39.156.125.20	SSL Continuation Data
39.156.125.20	10.128.205.212	SSL Continuation Data
10.128.205.212	39.156.125.20	TCP 62790 → 443 [ACK] Seq=55 Ack=75 Win=513 Len=0
10.128.205.212	10.3.9.44	DNS Standard query 0xc9d1 AAAA data.bilibili.com
10.3.9.44	10.128.205.212	DNS Standard query response 0xc9d1 AAAA data.bilibili.com CNAME data.bilibili.com
10.128.205.212	111.62.56.172	TCP 51100 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
111.62.56.172	10.128.205.212	TCP 443 → 51100 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1386 WS=512
10.128.205.212	111.62.56.172	TCP 51100 → 443 [ACK] Seq=1 Ack=1 Win=131584 Len=0
10.128.205.212	111.62.56.172	TLSv1.2 Client Hello
111.62.56.172	10.128.205.212	TCP 443 → 51100 [ACK] Seq=1 Ack=518 Win=30720 Len=0
111.62.56.172	10.128.205.212	TLSv1.2 Server Hello, Change Cipher Spec, Encrypted Handshake Message
10.128.205.212	111.62.56.172	TLSv1.2 Change Cipher Spec, Encrypted Handshake Message
10.128.205.212	111.62.56.172	TLSv1.2 Application Data
10.128.205.212	111.62.56.172	TLSv1.2 Application Data
10.128.205.212	111.62.56.172	TLSv1.2 Application Data
111.62.56.172	10.128.205.212	TCP 443 → 51100 [ACK] Seq=216 Ack=2054 Win=33280 Len=0
111.62.56.172	10.128.205.212	TLSv1.2 Application Data, Application Data, Application Data
10.128.205.212	111.62.56.172	TCP 51100 → 443 [ACK] Seq=2254 Ack=661 Win=130816 Len=0
10.128.205.212	10.3.9.44	DNS Standard query 0x6314 A s1.hdslb.com
10.128.205.212	10.3.9.44	DNS Standard query 0x2dc3 AAAA s1.hdslb.com
10.3.9.44	10.128.205.212	DNS Standard query response 0x6314 A s1.hdslb.com CNAME bstatic.hdslb.com CNAME
10.3.9.44	10.128.205.212	DNS Standard query response 0x2dc3 AAAA s1.hdslb.com CNAME bstatic.hdslb.com CNAME
10.128.205.212	111.42.184.216	TCP 51101 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
111.42.184.216	10.128.205.212	TCP 443 → 51101 [SYN, ACK] Seq=0 Ack=1 Win=42340 Len=0 MSS=1386 SACK_PERM=1 WS=
10.128.205.212	111.42.184.216	TCP 51101 → 443 [ACK] Seq=1 Ack=1 Win=131584 Len=0
10.128.205.212	111.42.184.216	TLSv1.2 Client Hello
111.42.184.216	10.128.205.212	TCP 443 → 51101 [ACK] Seq=1 Ack=598 Win=41984 Len=0
111.42.184.216	10.128.205.212	TLSv1.2 Server Hello
111.42.184.216	10.128.205.212	TLSv1.2 [TCP Previous segment not captured], Ignored Unknown Record
10.128.205.212	111.42.184.216	TCP 51101 → 443 [ACK] Seq=598 Ack=2773 Win=131584 Len=0 SLE=4159 SRE=4258
111.42.184.216	10.128.205.212	TCP [TCP Retransmission] 443 → 51101 [ACK] Seq=2773 Ack=598 Win=42496 Len=1386
10.128.205.212	111.42.184.216	TCP 51101 → 443 [ACK] Seq=598 Ack=4258 Win=131584 Len=0
111.30.159.64	10.128.205.212	OICQ OICQ Protocol
10.128.205.212	111.42.184.216	TLSv1.2 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
10.128.205.212	111.42.184.216	TLSv1.2 Application Data
10.128.205.212	111.42.184.216	TLSv1.2 Application Data

图 6-1：测试数据（节选）

保存抓包文件为 `test.pcapng`，并通过 WinSCP 软件，选择 SCP 协议发送至 OpenWrt 虚拟机，使其与 `fn_offline` 位于同一文件夹下。

```
root@OpenWrt:/featureG# ls
fn_live      fn_offline   test.pcapng
```

图 6-2：测试数据位置

执行以下命令运行可执行程序，并将输出重定位至 offline.log 文件下：

```
root@OpenWrt:/featureG# ./fn_offline test.pcapng >offline.log_
```

图 6-3：运行程序命令

在这一步中，通过命令行将输入文件的文件名 test.pcapng 传入离线流量特征器。执行完毕后，将生成的文本文件通过 SCP 协议传输至 Windows 主机查阅：

```
2182 Packet length: 5598
2183 Number of bytes: 5598
2184 Recieved time: Fri May 13 11:10:46 2022
2185 Ethernet Type:2048
2186 IP source addr:111.42.184.216
2187 IP destination addr: 10.128.205.212
2188 TCP source port: 443
2189 TCP destination port: 51101
2190 Forward packet add to exist flow
2191 This Segment:
2192 1652440246.458808,0.000000,0
2193 Segment Feature:
2194 0.852480,0.000000,0.000000,0.852480,0.000000,0.852480,0.196294,0.045199
2195 Flow Key:111.42.184.216,10.128.205.212,443,51101,Forward
2196 Time:1652440246.689070
2197 Delay:0.999874
2198 Forward
2199 Feature:48.007598,105182.778081,314026001.661808,34.007598,16524402.809960,27305587677324420.000000,48.007598,398
4.630622,330724.341651,0.346576,303944.899449,141.383141
Flow Feature:49,0,1.000000
```

图 6-4：特征提取结果（节选）

可以发现，这是一个携带段请求的 TCP 报文，因此输出了上一段的特征。

6.3.2 实时流量特征器的测试

经过本文第二章介绍的 OpenWrt 虚拟机网络配置方式后，开机状态的 OpenWrt 虚拟机可以做为主机的路由器。下面对此进行检验。

保持虚拟机处于开机状态，首先分别在 windows 主机和虚拟机上通过 ping 命令互相发送 ICMP 报文。本此测试中，主机地址为 192.168.126.100，虚拟机地址为 192.168.126.99。结果如图 6-5，图 6-6。

```
root@OpenWrt:/featureG# ping 192.168.126.100
PING 192.168.126.100 (192.168.126.100): 56 data bytes
64 bytes from 192.168.126.100: seq=0 ttl=128 time=0.520 ms
64 bytes from 192.168.126.100: seq=1 ttl=128 time=0.982 ms
64 bytes from 192.168.126.100: seq=2 ttl=128 time=1.059 ms
64 bytes from 192.168.126.100: seq=3 ttl=128 time=0.728 ms
^C
--- 192.168.126.100 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.520/0.822/1.059 ms
```

图 6-5：虚拟机 ping 主机

```
C:\Users\SPY>ping 192.168.126.99

正在 Ping 192.168.126.99 具有 32 字节的数据:
来自 192.168.126.99 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.126.99 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.126.99 的回复: 字节=32 时间<1ms TTL=64
来自 192.168.126.99 的回复: 字节=32 时间<1ms TTL=64

192.168.126.99 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

图 6-6: 主机 ping 虚拟机

若两方均无法 ping 通, 或其中一方无法 ping 通另一方, 应首先关闭主机和虚拟机防火墙: 在主机中修改防火墙设置, 在虚拟机中执行命令/etc/init.d/firewall stop。若仍无法成功, 则应检查虚拟机通过 NAT 模式上网所用的虚拟网卡 8 是否配置在它应所属的子网下。若否, 则应设置其 IP 地址为手动设置模式并修改 IP 地址。

在主机和虚拟机可以互通后, 检验虚拟机是否可以作为主机的路由器使用:

打开 wireshark, 选择网卡 Vmware Network Adapter Vmnet2 开始抓包。在主机上浏览网页, 观察抓包结果:

fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x2172 A optimizationguide-pa.googleap
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0xf443 AAAA optimizationguide-pa.googl
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x2172 A optimizationguide-pa
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0xf443 AAAA optimizationguide
fe80::20c:29ff:fe9...	fdcd:65b5:ddbc:0:bcd2:66be:...	ICMPv6	Neighbor Solicitation for fdcd:65b5:ddbc:0:bcd2:66be:...
fdcd:65b5:ddbc:0:b...	fe80::20c:29ff:fe99:4ca6	ICMPv6	Neighbor Advertisement fdcd:65b5:ddbc:0:bcd2:66be:ad5
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x440a A blog.csdn.net
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0xf76d A adservice.google.com
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0xc7ac AAAA blog.csdn.net
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0xae70 AAAA adservice.google.com
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x4244 A g.csdnimg.cn
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x1caa AAAA g.csdnimg.cn
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x440a A blog.csdn.net A 182.
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0xf76d A adservice.google.com
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0xc7ac AAAA blog.csdn.net SOA
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0xae70 AAAA adservice.google.
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x4244 A g.csdnimg.cn CNAME g
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x1caa AAAA g.csdnimg.cn CNAM
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x4def A csdnimg.cn
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x5911 AAAA csdnimg.cn
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x34f1 A dup.baidustatic.com
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x76ca A event.csdn.net
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0x855c AAAA dup.baidustatic.com
fdcd:65b5:ddbc:0:b...	fdcd:65b5:ddbc::1	DNS	Standard query 0xf0ad AAAA event.csdn.net
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x4def A csdnimg.cn CNAME csd
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x5911 AAAA csdnimg.cn CNAME
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x34f1 A dup.baidustatic.com
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x76ca A event.csdn.net CNAME
fdcd:65b5:ddbc::1	fdcd:65b5:ddbc:0:bcd2:66be:...	DNS	Standard query response 0x855c AAAA dup.baidustatic.c

图 6-7: 虚拟网卡 2 抓包结果 (节选)

观察发现该接口转发了大量 DNS 和 ICMP 报文。说明 OpenWrt 路由器已经作为主机的路由器开始工作。

下面开始动态流量特征器的测试, 在虚拟机中执行命令运行实时流量特征器, 并将输出重定位至文件 live.log 中:

```
root@OpenWrt:/featureG# ./fn_live>live.log
```

图 6-8: 运行在线流量特征器命令

打开视频网站 www.bilibili.com 观看视频, 同时观察流量特征器的运行状态。

20 分钟后，关闭视频网站，将生成的文本文件通过 SCP 协议发送至主机查看：

```
Packet length: 217
Number of bytes: 217
Recieved time: Fri May 13 12:08:35 2022
IP source addr: 192.168.126.100
IP destination addr: 239.255.255.250
UDP source port: 55405
UDP destination port: 1900
Forward packet add to exist flow
Flow Key: 192.168.126.100,239.255.255.250,55405,1900,Forward
Time: 1652443715.264473
Delay: 0.010000
Forward
Feature: 0.010000,2.170000,470.890000,0.010000,16524437.152645,27305702321170544.000000,0.000000,0.000000,0.0000
00,0.000000,0.000000,0.000000
Flow Feature: 0,0,0.000000
```

图 6-9：特征生成结果（节选）

期间，程序并未发生错误，说明本研究设计的轻量级流量特征器可以稳定地运行于路由器，并完成实时流量特征提取。

6.4 数据分析

为验证本研究设计的流量特征对 QoE 预测有现实意义，本研究提供了视频流量判断和分辨率预测功能。下面通过大量数据进行验证。本研究从提供视频服务的网站合计抓取来自不同分辨率，不同长度视频的 239MB 抓包文件作为原始数据进行处理，分析与学习。

出于效率和实时性考量，本研究设计实现的流量特征器并未在主体部分进行每种特征全部统计数据的计算，而选择计算并保存可用于计算出如均值，方差等统计特征的原始数据。为了进行分辨率预测，需要针对某些特征计算其具体统计数据。因此在原有特征计算模块的基础上，不改变现有结构，新增统计数据数据计算部分，用于计算以下特征并额外输出：

- 1) 包长的平均值，方差
- 2) 到达间隔的平均值，方差（放大 100 倍）
- 3) 带宽
- 4) 数据包吞吐量

此外，为了便于数据处理，本研究在进行本步骤测试时重新设计特征输出功能，使其输出为 CSV 文件，表头为：包长均值，包长方差，间隔均值，间隔方差，带宽，吞吐量，预测值。

```

1 aS,vS,I,aI,vI,throughput,pps,resolu
2 1440.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,
3 1440.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,
4 1440.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,
5 1440.000000,0.000000,0.000000,0.000000,0.000000,0.000000,0.000000,
6 1440.000000,0.000000,0.150490,0.030121,0.000000,956875.437262,3322.484157,
7 1440.000000,0.000000,0.000000,0.025094,0.000000,1913750.874525,3986.980989,
8 1440.000000,0.000000,0.000000,0.021502,0.000000,2870626.311787,4651.477820,
9 1440.000000,0.000000,0.000000,0.018807,0.000000,3827501.749049,5315.974651,
10 1440.000000,0.000000,0.442004,0.065973,0.000000,1215202.156855,1519.002696,
11 1440.000000,0.000000,0.000000,0.059353,0.000000,1458242.588226,1687.780773,
12 1440.000000,0.000000,0.000000,0.053936,0.000000,1701283.019597,1856.558851,
13 1440.000000,0.000000,0.000000,0.049422,0.000000,1944323.450968,2025.336928,
14 1440.000000,0.000000,0.000000,0.045603,0.000000,2187363.882339,2194.115005,
15 1440.000000,0.000000,0.000000,0.042329,0.000000,2430404.313710,2362.893083,
16 1440.000000,0.000000,0.000000,0.039491,0.000000,2673444.745081,2531.671160,
17 1440.000000,0.000000,0.000000,0.037009,0.000000,2916485.176452,2700.449237,
18 1440.000000,0.000000,0.000000,0.034818,0.000000,3159525.607823,2869.227315,
19 1440.000000,0.000000,0.000000,0.032871,0.000000,3402566.039194,3038.005392,
20 1440.000000,0.000000,0.000000,0.031129,0.000000,3645606.470565,3206.783469,
21 1440.000000,0.000000,0.000000,0.029561,0.000000,3888646.901936,3375.561547,
22 1440.000000,0.000000,0.000000,0.028142,0.000000,4131687.333307,3544.339624,
23 1440.000000,0.000000,0.000000,0.026852,0.000000,4374727.764677,3713.117702,

```

图 6-10 特征器输出示例

将特征输出组织为 CSV 文件后，需要将来自同一分辨率的特征文件组织为一个表格。同时需要删除部分冗余数据。编写 python 程序实现此功能：

```

1 import pandas as pd
2 import os
3 df_list = []
4 #合并CSV文件
5 for i in os.listdir():
6     if "1080P" in i:
7         fn = "1080P"
8         df=pd.read_csv(i)
9         df_list.append(df)
10 df = pd.concat(df_list, axis=0)
11 #删除预测值为空的行
12 df.dropna(axis=0, how='all', subset=['resolu'], inplace=True)
13 df.to_csv("1080P.csv", index=0)

```

图 6-11 合并 csv 文件的 python 代码

合并后的文件及文件大小如下：





 360P.csv	2022/5/24 15:04	Microsoft Exc...	926 KB
 480P.csv	2022/5/24 15:05	Microsoft Exc...	3,917 KB
 720P.csv	2022/5/24 15:05	Microsoft Exc...	2,308 KB
 1080P.csv	2022/5/24 15:06	Microsoft Exc...	5,454 KB

图 6-12 合并后文件展示

下面对特征数据进行分析学习，统计带宽和数据包吞吐量的均值，以及三倍标准差内的数据集合，并观察延迟数据分布的特点。编写 python 程序进行分析。

```
1 import pandas as pd
2 fn = "360P.csv"
3 df = pd.read_csv(fn, encoding='utf-8')
4 mean_bw = df["throughput"].mean()
5 std_bw = df["throughput"].std()
6 mean_tp = df["pps"].mean()
7 std_tp = df["pps"].std()
8
9 print("mean bandwidth:{0},std bandwidth:{1},mean throughput:{2},std throughput:{3}".format(mean_bw,std_bw,mean_tp,std_tp))
10 print("good bandwidth:{0} to {1}".format(mean_bw-2*std_bw,mean_bw+2*std_bw))
11 print("good throughput:{0} to {1}".format(mean_tp-2*std_tp,mean_tp+2*std_tp))
```

图 6-13 数据分析代码

在分析过程中，发现少量极端数据会导致结果波动极大。因此判断存在损坏，或错误抓取的数据。故排除首先排除带宽和吞吐量在三倍标准差外的特征结果，再进行数据分析。分析结果(含舍入)如下，其中 good 数据表示均值加减两倍标准差范围内的数据。

1) 360P:

mean bandwidth:28864

std bandwidth:4198

mean throughput:20.39

std throughput:2.92

good bandwidth:20467 to 37260

good throughput:14 to 26

2) 480P:

mean bandwidth:45704

std bandwidth:15757

mean throughput:32.35

std throughput:11.13

good bandwidth:14188 to 77219

good throughput:10 to 54

3) 720P:

mean bandwidth:182606

std bandwidth:64047

mean throughput:128.40

std throughput:45.42

good bandwidth:54510 to 310701

good throughput:37 to 219

4) 1080P 及以上:

mean bandwidth:320093

std bandwidth:174784


```
mean throughput:247.16
std throughput:131.40
good bandwidth:-29475 to 669662
good throughput:-15 to 509
```

对于 1080P 数据，由于采集时间长，受干扰较大，样本的标准差较大。因此重新选择均值加减一倍标准差范围作为 good 数据。

5) 1080P 及以上（一倍标准差）:

```
mean bandwidth:320093
std bandwidth:174784
mean throughput:247.16
std throughput:131.40
good bandwidth:145309 to 494878
good throughput:115 to 378
```

初步分析发现，来自各分辨率抓包结果的统计特征重合有重合。进一步分析重合部分特征。首先考察 360P 数据和 480P 数据在带宽重叠的部分吞吐量是否存在差异，分别计算两种数据在重叠部分吞吐量的均值加减一倍标准差范围，代码如下：

```
1 import pandas as pd
2 fn = "360P.csv"
3 df = pd.read_csv(fn,encoding='utf-8')
4 mean_bw = df["throughput"].mean()
5 std_bw = df["throughput"].std()
6 df_overlap = df[(14188< df.throughput) &(df.throughput< 37260 ) ]
7
8 mean = df_overlap.pps.mean()
9 std = df_overlap.pps.std()
10 print(mean-std,mean+std)
```

图 6-14 数据分析代码 2

得到 360P 数据吞吐量范围为（18.73， 21.00），480P 数据吞吐量范围为（20.51， 24.24）。发现重叠部分缩小。分别对其它分辨率结果进行类似分析，结果为：

```
14188~37260, 360P (18.73, 21.00) 480P (20.51, 24.24)
54510~77219, 480P (39.79, 48.99) 720P (50.98, 56.56)
77219~145309 720P (61.69, 84.99) 1080P (93.81, 114.31)
145309~310701, 720P (127.33, 154.93) 1080P (168.36, 218.32)
```

可以看出，不同分辨率视频流量的特征数据在带宽重叠的部分吞吐量并不重叠，由这两种特征可以实现视频流量的 QoE 预测。拟定分类标准如下：

带宽	吞吐量	预测值
14188以下	无要求	360P
14188-37260	21以下	360P
14188-37260	21以上	480P
37260-54510	无要求	480P
54510-77219	50以下	480P
54510-77219	50以上	720P
77219-145309	89以下	720P
77219-145309	89以上	1080P以上
145309-310701	160以下	720P
145309-310701	160以上	1080P以上
310701以上	无要求	1080P以上

图 6-15 分辨率预测标准

下面使用分类标准对抓包结果进行分辨率预测。根据上述逻辑编写代码用于在特征提取后判断分辨率：

```

if (this->is_vedio){
//printf("P:");
if(this->bwd_flow_bps<14188) printf("360\n");
else if (this->bwd_flow_bps<37260 && this->bwd_flow_pps<21) printf("360\n");
else if (this->bwd_flow_bps<37260 && this->bwd_flow_pps>=21) printf("480\n");
else if (this->bwd_flow_bps>37260 && this->bwd_flow_bps<54510) printf("480\n");
else if (this->bwd_flow_bps>54510 && this->bwd_flow_bps<77219 && this->bwd_flow_pps<50) printf("480\n");
else if (this->bwd_flow_bps>54510 && this->bwd_flow_bps<77219 && this->bwd_flow_pps>=50) printf("720\n");
else if (this->bwd_flow_bps>77219 && this->bwd_flow_bps<145309 && this->bwd_flow_pps<89) printf("720\n");
else if (this->bwd_flow_bps>77219 && this->bwd_flow_bps<145309 && this->bwd_flow_pps>=89) printf("1080\n");
else if (this->bwd_flow_bps>145309 && this->bwd_flow_bps<310701 && this->bwd_flow_pps<160) printf("720\n");
else if (this->bwd_flow_bps>145309 && this->bwd_flow_bps<310701 && this->bwd_flow_pps>=160) printf("1080\n");
else printf("1080\n");
}
else printf("\n");

```

图 6-16 判断分辨率代码

产生特征提取与分辨率预测结果后，重复上述文件合并步骤将其按照分辨率合并为 4 个文件，名称为“分辨率 total.csv”。使用如下代码对各个分辨率的预测结果进行统计：

```

1 import pandas as pd
2 fn = "360total.csv"
3 df = pd.read_csv(fn,encoding='utf-8')
4 c1,c2,c3,c4=0,0,0,0
5 for i in df.resolu:
6     if i == 360:
7         c1 = c1+1
8     elif i==480:
9         c2 = c2+1
10    elif i == 720:
11        c3 = c3+1
12    elif i ==1080:
13        c4 = c4+1
14    c = c1+c2+c3+c4
15    print(c1/c,c2/c,c3/c,c4/c)

```

图 6-17 结果统计代码

最终预测结果及准确率如下：

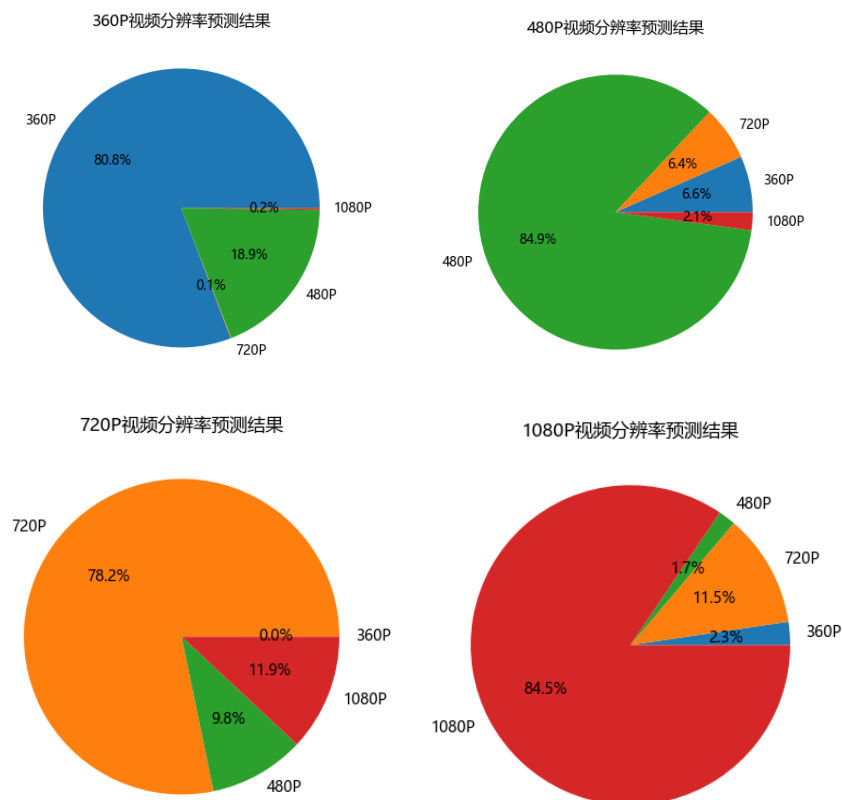


图 6-18 最终预测结果

且综合准确率达到 82.21%。

6.5 本章小结

本章介绍轻量级流量特征器的部署方法，使用流量数据分别测试了离线与实时流量特征器。最后收集数据对所提取的流量特征进行验证。证明了轻量级流量特征器的可行性与稳定性，以及特征提取的有效性。

第七章 总结与展望

7.1 总结

7.1.1 系统实现的意义

现如今，视频服务的 QoE 预测已经成为一个热点话题。准确地实时预测用户体验质量有利于网络运营商针对链路状态采取及时的应对措施，进而改善用户 QoE，提高用户忠诚度。流量特征工程对 QoE 预测有着至关重要的意义，目前几乎所有 QoE 预测都需要这一步工作。

然而随着流量加密技术的普及，如今绝大多数视频服务提供者都选择了各种加密协议来保证流量的加密传输，以往透明的应用标签受到隐藏，这对流量特征的提取带来极大的挑战。流量特征器无法在大量数据包中识别出属于需要预测 QoE 的服务的流量，这使得后续分类器的设计变得复杂困难，且准确率会大打折扣。

本研究设计的轻量级加密流量特征器最大的优势在于轻量级。轻量级意味着更稳定的系统及更高的实时性。在此基础上，本研究针对视频服务 QoE 指标的半段方式设计流量特征集合，保证了特征提取的有效性，可以简化后续分类器的设计难度，并提高准确率。

7.1.2 系统已完成的工作

本研究设计的轻量级流量特征器完成了数据包的抓取，分析；流量管理以及流量特征的计算、输出。完成了系统所需动态库的配置与交叉编译。可以实现多平台上流量特征的生成工作。

此外，本研究选取 OpenWrt 开源路由器，并将流量特征器成功在 OpenWrt 路由器上部署并运行。

7.1.3 系统的改进方向

首先，本研究设计的流量特征器依赖于网络层和应用层首部信息，因此仅适用于获取经 HTTPS 和 QUIC 协议加密的数据流量的特征，而不适合在传输层及以下层次加密的协议，如 TLS 和 Ipsec。

其次，在信息处理层面，本研究并未提供对 IPv6 等协议的支持。在后续的工作中，应进一步增加对更多协议，更多种类协议栈的支持，力求全方位地捕获网络流量并分析，借此更加有效地实现 QoE 预测。

最后，在兼容性方面，本研究只能基于 linux 内核且可以运行 gcc 的操作系统中运行，且需要复杂的交叉编译。若要提供对 windows 系统的支持，则需要修改源代码，将其中调用 libpcap 动态库函数的部分改为调用 Winpcap 库中功能类似的函数。

7.2 展望

纵观这十年来国内外视频网站的兴衰，不难看出长视频正日渐式微，而短视频这颗新星正在冉冉升起。但无论如何——视频——已经占据了人们日常生活中不可或缺的一席。繁多的视频平台在互联网的大背景下展开了激烈的竞争，谁能得到用户，谁能留住用户，谁就能立于不败之地。而保持用户粘性的除了运营手段，用户体验质量也是至关重要的一点。本研究正是立足用户体验质量的准确实时预测这一实际需求而诞生的。

在 8 年之前，互联网上仅有 50% 的流量是加密的。Google 透明度报告则显示，2016 年 chrome 加载的网页中启用加密的比例达到 75%，而这一数据在 2019 年已达到 95%。几乎所有网络服务均已启用各种不同的加密协议。在未来，这一比例甚至还会进一步提高，直到在互联网上加密流量完全取代未经加密的流量。这样的环境之下，本研究设计的加密流量特征器的作用将越来越大。此外，由于本研究设计的特征器具有较高的可扩展性，即使技术更新，加密协议更新换代，也可以基于所用的路由器平台进行尽可能地升级与拓展。

7.3 心得与收获

通过本论文的研究，我对流量特征工程以及视频服务 QoE 预测有了更深刻的认识。与此同时，本研究对我的个人能力起到了全方位的促进。在本次研究中，随着研究进程的发展，我逐渐习得并强化了查找文献，从文献中获取关键信息的能力；linux 系统下使用 makefile 搭建项目，使用 gdb 和 strace 命令调试的能力。在跨环境编译过程的不断试错中，我自主发现问题，自主研究解决问题的能力也得到强化。

此外，在 Ubuntu 和 OpenWrt 这两个不熟悉的操作系统上工作时遇到的种种问题也促使我研究类 linux 系统的内核机制与系统调用，极大加强了我对操作系统相关知识的了解。作为一名计算机科学与技术专业的学生，专业知识的学习、运用使我感到满足而愉快。此次研究也培养了我终身学习的能力。我希望在今后的工作学习中进一步学习、利用信息网络相关的知识，参与到打造现代信息化强国的浪潮之中。

参考文献

- [1] Liu C , He L , Xiong G , et al. FS-Net: A Flow Sequence Network For Encrypted Flow Classification[C]// IEEE INFOCOM 2019 - IEEE Conference on Computer Communications. IEEE, 2019.
- [2] A.W. Moore, D. Zuev. Internet traffic classification using bayesian analysis techniques[C].In Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Banff, 2005, 50 — 60.
- [3] Mazhar M H, Shafiq Z. Real-time video quality of experience monitoring for https and quic[C]//IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2018: 1331-1339.
- [4] Bronzino F, Schmitt P, Ayoubi S, et al. Inferring streaming video quality from encrypted flow: Practical models and deployment experience[J]. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 2019, 3(3): 1-25.
- [5] 季庆光, 冯登国. 对几类重要网络安全协议形式模型的分析[J]. 计算机学报, 2005, 28(7):13.
- [6] libpcap&tcpdump documentation. <https://www.tcpdump.org/>
- [7] i-derry. OpenWrt 内核模块开发(九)-通过 linux netfilter 框架实现数据包分析. https://blog.csdn.net/dxt1107/article/details/118891952?share_token=919a529c-b22a-4e1b-9bff-55bb0ea35d6a
- [8] Mangla T , Halepovic E , Ammar M , et al. Using Session Modeling to Estimate HTTP-based Video QoE Metrics from Encrypted Network Flow[J]. IEEE Transactions on Network and Service Management, 2019, PP(3):1-1.
- [9] develop a packet sniffer with libpcap. <https://vichargrave.github.io/programming/develop-a-packet-sniffer-with-libpcap/>
- [10] use libpcap in C. <https://www.devdungeon.com/content/using-libpcap-c>
- [11] naturebe. pcap 结构—pcap_t 数据类型. <https://blog.csdn.net/naturebe/article/details/6881290>
- [12] yuanfan2012. Linux 下网卡的混杂模式浅谈. <https://cloud.tencent.com/developer/article/1439013>
- [13] OpenBSD PF - User's Guide. <https://www.openbsd.org/faq/pf/>
- [14] tcspecial. libpcap 交叉编译. <https://www.iteye.com/blog/tcspecial-2323731>

致 谢

感谢毕设期间张沛老师与张晗老师对我的指导和帮助。感谢 OpenWrt sdk 的提供者极大简化了本研究完成软件部署的难度。感谢家人和北京邮电大学在毕设期间为我提供住所。感谢同学和室友的支持与帮扶。

附 录

附录1 flow 类完整定义及构造函数

```
class flow
{
public:
    flow(struct simple_packet_info &simple_packet_info);

    struct flow_key flow_key;

    int reverse, is_vedio, is_judged;

    u_int8_t ip_protocol;

    time_type fwd_latest_timestamp;

    std::vector<packet_count_type> fwd_last_TCPseq;

    packet_count_type raw_fwd_packet_size_ls;

    time_type fwd_interarrival_time_n, fwd_interarrival_time_ls,
fwd_interarrival_time_ss;

    double fwd_packet_size_n, fwd_packet_size_ls, fwd_packet_size_ss;

    double fwd_window_size_n, fwd_window_size_ls, fwd_window_size_ss;

    double fwd_flow_bps, fwd_flow_pps;

    time_type fwd_flow_duration;

    time_type fwd_start_timestamp;

    //backward flow character

    time_type bwd_latest_timestamp;

    std::vector<packet_count_type> bwd_last_TCPseq;

    packet_count_type raw_bwd_packet_size_ls;

    time_type bwd_interarrival_time_n, raw_bwd_interarrival_time_ls,
bwd_interarrival_time_ls, bwd_interarrival_time_ss;

    double bwd_packet_size_n, bwd_packet_size_ls, bwd_packet_size_ss;

    double bwd_window_size_n, bwd_window_size_ls, bwd_window_size_ss;

    double bwd_flow_bps, bwd_flow_pps;

    time_type bwd_flow_duration;

    time_type bwd_start_timestamp;

    //segment character
```

```
packet_count_type segment_count, segment_size_temp;
double segment_size_n, segment_size_ls, segment_size_ss;
time_type          segment_req_interval_n,          segment_req_interval_ls,
segment_req_interval_ss;
time_type          last_req_time,
segment_duration_temp, segment_duration_n, segment_duration_ls, segment_duration_ss;

//flow character related

double flow_tcp_rate;
packet_count_type retrans;
packet_count_type tcp_count, fwd_pkt_count, bwd_pkt_count;
packet_count_type tcp_urgent_count;

void add_packet(simple_packet_info &pkt);

double decay(double decay_factor ,simple_packet_info &pkt);
void output_packet_c(simple_packet_info &pkt, double decay);
void terminate();
void to_reverse();
void judge_vedio();
};

flow::flow(simple_packet_info& simple_packet_info)
{

    this->flow_key = simple_packet_info.flow_key;

    this->fwd_start_timestamp = simple_packet_info.ts;
    this->reverse = 0;
    this->is_vedio=0;
    this->is_judged=0;
    this->fwd_interarrival_time_n = 0;
```



```
this->fwd_window_size_n = 0;
this->fwd_packet_size_n = 0;
this->fwd_interarrival_time_ls = 0;
this->fwd_interarrival_time_ss = 0;
this->fwd_window_size_ls = 0;
this->fwd_window_size_ss = 0;
this->fwd_packet_size_ls = 0;
this->raw_fwd_packet_size_ls = 0;
this->fwd_packet_size_ss = 0;

this->bwd_start_timestamp = simple_packet_info.ts;
this->bwd_interarrival_time_n = 0;
this->bwd_window_size_n = 0;
this->bwd_packet_size_n = 0;
this->bwd_interarrival_time_ls = 0;
this->raw_bwd_interarrival_time_ls = 0;
this->bwd_interarrival_time_ss = 0;
this->bwd_window_size_ls = 0;
this->bwd_window_size_ss = 0;
this->bwd_packet_size_ls = 0;
this->raw_bwd_packet_size_ls = 0;
this->bwd_packet_size_ss = 0;

this->tcp_count = 0;
this->fwd_pkt_count = 0;
this->bwd_pkt_count = 0;
this->retrans = 0;
this->tcp_urgent_count = 0;

this->segment_count = 0;
this->segment_req_interval_n = 0;
this->segment_req_interval_ls = 0;
this->segment_req_interval_ss = 0;
this->segment_size_temp = 0;
this->segment_size_n = 0;
this->segment_size_ls = 0;
```

```
this->segment_size_ss = 0;
this->segment_duration_temp = 0;
this->segment_duration_n = 0;
this->segment_duration_ls = 0;
this->segment_duration_ss = 0;
this->last_req_time = 0;

}
```

