Netflow 项目说明文档

许祖耀-2120250722

计算机网络技术03521705-课程作业

2025 年 11 月 30 日

摘要

本报告详细阐述了 NetFlow —— 模块化网络流量分析与可视化平台的设计与实现。该项目旨在构建一个跨平台、可扩展的桌面应用，基于 Electron 框架开发，通过多进程架构整合了 Node.js 的系统能力与 Chromium 的渲染优势。核心功能包括基于 Python Raw Socket 的底层数据包捕获与动态过滤引擎、采用沙箱机制保障安全性的插件化生态系统、以及支持高性能虚拟滚动与 Canvas 绘图的可视化组件。此外，项目还实现了一个灵活的 Docking 停靠系统，允许用户自定义多窗口工作区。报告将深入探讨从底层抓包、IPC 进程间通信、数据管线分发到前端高性能渲染的完整技术链路，展示在计算机网络与现代软件工程领域的综合实践能力。

目录

1 项目结构与工程一览

Netflow是一个模块化的网络流量分析与可视化工具，是基于 Electron 框架实现的跨平台桌面应用，其支持插件化扩展抓包、数据处理与可视化功能。 Electron 提供主进程（Node.js 环境）与渲染进程（Chromium 环境）的多进程架构，实现跨平台GUI与本地系统交互。具体功能如抓包、数据解析与可视化均通过插件实现，插件以声明式 JSON 格式注册，运行在受限的沙箱环境中，确保安全性与稳定性。

下面是Netflow的关键文件及其职责：

路径	说明
`main.js`	Electron 主进程入口：初始化应用、插件管理器与主 IPC 通道。
`preload-base.js`, `preload-generated.js`	Preload 脚本：在渲染进程安全地暴露受限 API（插件沙箱权限边界）。
`renderer/`	主渲染层代码（UI 视图、Docking 管理、插件渲染容器）。
`utils/ipc.js`	主/渲染进程间与插件间通信的抽象层（message contracts、版本兼容）。
`plugins/`	插件目录：每个插件为独立文件夹，包含 `plugin.json`、主入口与 UI 资源。
`plugins/*/plugin.json`	插件声明：类型（sniffer/visualizer/reporter）、入口脚本、UI 桌面/Popout 配置、权限说明。
`plugins/python-sniffer/`	Sniffer 插件示例（通过外部进程与主进程通信以获取原始包数据）。
`plugins/packet-visualizer/`	数据可视化插件：订阅解析后流量并渲染图表/拓扑。
`docs/`	说明文档。

1.1 插件接口

每个插件遵循统一的 JSON 声明和运行时接口： plugin.json 示例：

```
{
  "name": "python-sniffer",
  "type": "sniffer",
  "entry": "index.js",
  "preload": "preload.js",
  "ui": "ui.html",
```

```
  "renderer": "renderer.js",
  "version": "1.0.0",
  "author": "netflow"
}
```

字段说明：

- `name`：插件唯一标识符。
- `type`：插件类型（sniffer/visualizer/reporter）。
- `entry`：主进程入口脚本路径。
- `preload`：渲染进程 Preload 脚本路径（暴露受限 API）。
- `ui`：插件 UI 主 HTML 文件路径。
- `renderer`：渲染进程入口脚本路径。
- `version`：插件版本号。
- `author`：插件作者信息。

Netflow 通过解析该 JSON 文件动态加载插件，并根据声明的入口脚本和 UI 资源初始化插件实例, 确保插件在受控环境中运行，具体实现详见第 3节。

1.2 插件总览

目前Netflow实现了以下插件：

- **Python Sniffer**：基于 Python 的抓包插件，启动外部进程捕获原始网络包并通过 Stdout 传输给主进程。
- **Packet Visualizer**：数据包可视化插件，订阅解析后的流量数据并以虚拟列表形式展示，支持过滤与搜索。
- **Traffic Grapher**：实时流量图表插件，将流量数据聚合到时间桶中并绘制折线图，支持多种协议分类。
- **Network Reporter**：网络报告生成器插件，基于捕获的数据生成 HTML 格式的流量分析报告。

其中，Python Sniffer 插件负责底层数据采集，其他插件则订阅数据并提供不同的可视化与分析功能。插件通过 IPC 与主进程通信，确保数据流的实时传输与处理。

2 关键功能实现（抓包与插件生态）

本节详细说明抓包模块、插件生态策略、数据管线与 Docking 系统的实现细节，便于教师评估设计的技术深度。

2.1 抓包实现（Python Sniffer）

抓包模块的核心是一个独立运行的 Python 脚本（`sniffer.py`），它通过原始套接字（Raw Socket）直接与网络接口交互。

2.1.1 底层捕获机制

使用了 Python 的 `socket` 库，配置为 `AF_INET` 和 `SOCK_RAW`，并设置 `IP_HDRINCL` 选项以包含 IP 头。在 Windows 平台上，还需要调用 `ioctl(SIO_RCVALL)` 以启用混杂模式，捕获流经网卡的所有数据包。

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.
    IPPROTO_IP)
self.socket.bind((self.local_ip, 0))
self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
```

Listing 1: Python Raw Socket 初始化

2.1.2 协议解析与过滤

捕获到的二进制数据包首先被解析 IP 头（前20字节），提取源/目的 IP 和协议类型。如果是 TCP/UDP，进一步解析端口号以推断应用层协议（如 HTTP, DNS）。过滤器实现了一个基于 `eval` 的动态求值引擎。用户输入的过滤表达式（如 `src 192.168.1.1 and proto TCP`）被转换为 Python 表达式，并在包含 `src_ip`, `dst_ip` 等变量的上下文中执行。这种方式提供了极高的灵活性，支持复杂的逻辑组合。

2.1.3 进程间通信 (IPC)

Python 进程将解析后的数据包以结构化的文本格式输出到标准输出（stdout）。Electron 主进程通过 Node.js 的 `child_process.spawn` 启动该脚本，并监听其 stdout 数据流，按行缓冲并解析为 JSON 对象，最后通过 IPC 广播给发送回渲染进程的嗅探器插件，实现数据的实时传输。嗅探器插件接收数据，进行必要的预处理后，将数据添加至DOM全局变量，并广播给所有注册的可视化插件。

2.2 数据流与广播（Data Pipeline）

数据从捕获到展示经历了以下阶段：

1. **采集层**：Python 进程捕获原始字节，解析出关键字段（五元组、长度、Payload），并进行初步过滤。
2. **传输层**：主进程通过 Stdout 接收文本流，反序列化为 JavaScript 对象，并附加时间戳。
3. **接收层**：主进程通过 `packet-data-batch` 频道将数据包数组广播发送到所有渲染器中的嗅探器插件。
4. **处理层**：嗅探器插件接收数据，进行预处理并存储于全局DOM变量，广播给所有注册的可视化插件。
5. **展示层**：各插件根据自身需求处理数据。例如，Visualizer 将数据存入内存数组并更新 UI，Grapher 将数据聚合到时间桶中。

下面的流程图展示了 Netflow 的多层架构与插件环境：

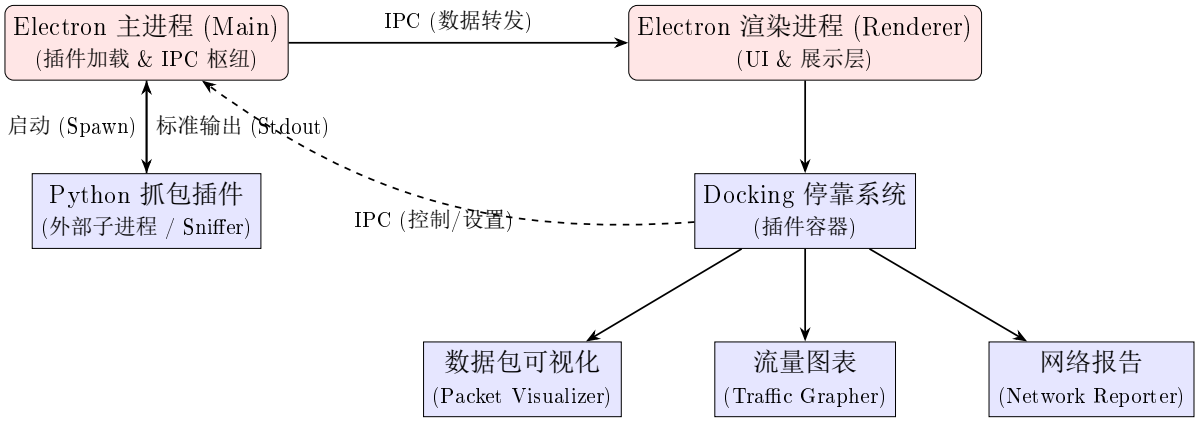

图 1: Netflow 架构图（进程分离与数据流向）

2.3 插件生态与安全边界

为保证稳定与可测性，插件运行在严格的Electron沙箱环境中。每个插件通过其 `plugin.json` 声明所需要加入的脚本代码，包括主进程入口、渲染进程 Preload 脚本和 UI 资源。Preload 脚本负责在渲染器中暴露受限的 API，允许插件与主进程通过 IPC 通信，但禁止直接访问 Node.js 或 DOM，防止恶意代码执行。

2.4 Docking 与 Popout 系统

项目实现了可停靠（docking）与弹出窗口（popout）能力，核心实现点：

- Docking 容器：渲染层提供一个 Docking 管理组件（`renderer/docking.js` + `docking.css`），支持面板的拖拽、停靠与拆分。
- Popout 窗口：插件可声明 `ui.popout = true`，由宿主通过 `BrowserWindow` 新建窗口加载该插件的 `ui.html`，并保留与主进程的双向 IPC。

3 插件系统设计与实现

NetFlow 采用高度模块化的架构，核心功能均通过插件实现。本节将详细阐述插件系统的底层实现机制，涵盖从插件发现、加载到前端隔离渲染的全过程。涉及的核心文件包括 `main.js`、`utils/ipc.js`、`renderer/app.js` 以及 `preload-base.js`。

3.1 插件发现与动态预加载 (Backend)

在 Electron 主进程启动时，系统通过 `utils/settings-loader.js` 执行插件发现逻辑：

1. **目录扫描**：遍历 `plugins/` 目录，读取每个子目录下的 `plugin.json` 清单文件，获取插件的元数据（名称、类型、入口文件等）。
2. **动态 Preload 组装**：为了赋予插件访问 Node.js 底层能力（如文件系统、网络）的同时保持上下文隔离，系统在启动时动态生成 `preload-generated.js`。
   - 系统首先读取基础预加载脚本 `preload-base.js`，其中定义了核心的 `netflowAPI`。
   - 接着遍历所有已启用的插件，将其定义的 `preload.js` 内容追加到文件中，并使用立即执行函数表达式 (IIFE) 包裹以避免全局变量污染。
   - 最终生成的脚本被设置为 `BrowserWindow` 的 `webPreferences.preload` 属性。

3.2 前端沙箱与隔离渲染 (Renderer)

在渲染进程 (`renderer/app.js`) 中，插件的 UI 和逻辑被严格隔离，以确保系统的稳定性和安全性。

为了防止插件的 CSS 样式污染全局或其他插件，系统为每个插件实例创建一个宿主容器，并开启 **Shadow DOM (Open Mode)**。插件的 HTML 内容 (`uiContent`) 直接注入到 Shadow Root 中，从而实现样式的完全隔离。因此插件的前端逻辑 (`renderJS`) 并非直接执行，而是经过特殊的封装处理：

1. **Blob URL 加载**：插件 JS 代码被转换为 Blob 对象，并通过 `<script src="blob:...">` 标签动态注入。
2. **作用域注入**：代码被包裹在一个闭包中，接收当前插件实例的 `scopeId`（宿主元素 ID）。
3. **DOM 操作代理 (Proxy)**：为了让插件开发者能像操作普通文档一样操作 Shadow DOM，系统创建了一个 `window.document` 的 **Proxy 代理对象**。

```
 const document = new Proxy(window.document, {
     get: (target, prop) => {
         // Intercept DOM query methods and redirect to ShadowRoot
         if (prop === 'getElementById') return (id) => shadowRoot.getElementById(id);
         if (prop === 'querySelector') return (selector) => shadowRoot.querySelector(selector);
         // ...
         return target[prop];
     }
 });

```

Listing 2: DOM 代理示例

通过这种机制，插件代码中的 `document.getElementById` 实际上是在该插件的 Shadow Root 范围内查找元素，从而实现了 DOM 操作的虚拟化和隔离。

3.3 IPC 通信桥接

主进程与渲染进程的通信通过 `utils/ipc.js` 和 `preload-base.js` 协同完成：

- **Context Bridge：** `preload-base.js` 使用 `contextBridge.exposeInMainWorld` 将安全的 API 暴露给 `window.netflowAPI`，阻断了渲染进程对 Node.js 环境的直接访问。
- **资源请求：** 当渲染进程需要加载插件时，通过 `invoke('get-plugins')` 向主进程请求。主进程读取插件的 HTML 和 JS 文件内容并返回，而不是让渲染进程直接读取文件系统。

4 内置插件功能详解

4.1 数据包可视化 (Packet Visualizer)

该插件提供了类似 Wireshark 的列表视图。为了处理大量数据包（可能达到数万条）导致的 DOM 性能问题，实现了**虚拟滚动 (Virtual Scrolling)**技术。

- **原理：** 仅渲染当前视口可见的行（加上少量缓冲区）。通过计算滚动条位置，动态切片数据数组并生成 HTML。
- **Hex 视图：** 点击数据包可查看十六进制转储（Hex Dump），同时显示对应的 ASCII 字符，便于分析 Payload 内容。

4.2 流量图表 (Traffic Grapher)

该插件实时展示网络流量趋势。

- **渲染技术：**使用 HTML5 `<canvas>` API 进行高性能绘图，避免了大量 DOM 节点操作。
- **数据聚合：**将接收到的数据包按秒聚合（Time Bucketing），计算每秒的字节数和包数。
- **交互：**支持鼠标悬停显示具体时刻的流量数值（Tooltip）。

4.3 网络报告 (Network Reporter)

该插件用于生成专业的 PDF 分析报告。

- **技术栈：**集成 `jspdf` 和 `jspdf-autotable` 库。
- **统计分析：**在前端计算协议分布（Protocol Distribution）、Top 5 源/目的 IP（Top Talkers）等统计信息。
- **自定义字体：**支持加载本地 TTF 字体文件，以解决 PDF 生成中的中文乱码问题。

5 程序流程

NetFlow 的运行流程涉及 Electron 主进程、渲染进程以及外部 Python 嗅探进程之间的紧密协作。

1. **系统初始化：**应用启动时，`main.js` 初始化 Electron 主进程，创建主窗口，并实例化 `PluginManager` 和 `DockingManager`。
2. **插件加载：**`PluginManager` 扫描 `plugins/` 目录，读取 `plugin.json` 配置，根据插件类型（Sniffer, Visualizer, Reporter）将其注册到系统中。
3. **界面渲染：**渲染进程加载 `index.html`，初始化 Docking 布局，并向主进程请求已加载的插件 UI 资源。
4. **嗅探器启动：**主进程通过 `child_process` 模块生成 Python 子进程 (`sniffer.py`)，用于底层数据包捕获。
5. **数据捕获与传输：**
    - Python 进程使用原始套接字 (Raw Socket) 捕获网络数据包。
    - 解析数据包头部信息，并通过标准输出 (Stdout) 发送。

- 主进程监听 Stdout，接收数据并通过 IPC 通道发送至嗅探器插件。
- 嗅探器插件接收数据，进行必要的预处理后，将数据添加至DOM全局变量，并广播给所有注册的可视化插件。

6. **可视化与交互：**各可视化插件接收数据流，更新内存中的数据结构，并驱动 Canvas 或 DOM 进行实时渲染。

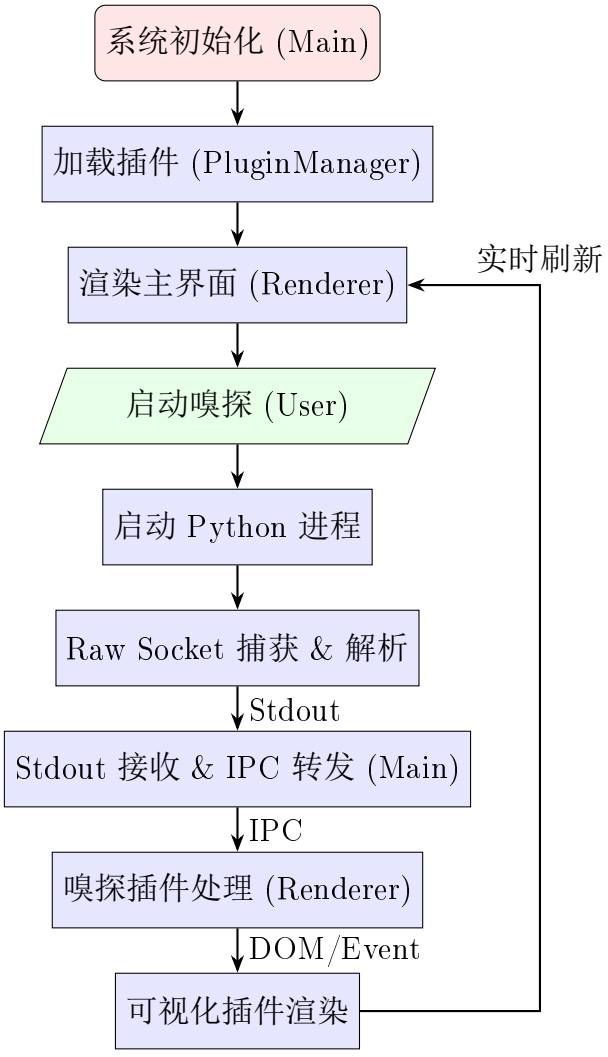

图 2: NetFlow 系统运行流程图

6 测试与截图

本节展示了 NetFlow 系统的实际运行效果，涵盖了从主界面布局到各个功能插件的具体操作视图。

6.1 主界面与 Docking 系统

图 3 展示了 NetFlow 的主界面，采用了灵活的 Docking 布局系统。用户可以自由拖拽、停靠各个插件窗口，构建个性化的工作区。图中展示了多个插件协同工作的场景，体现了多任务处理的能力。

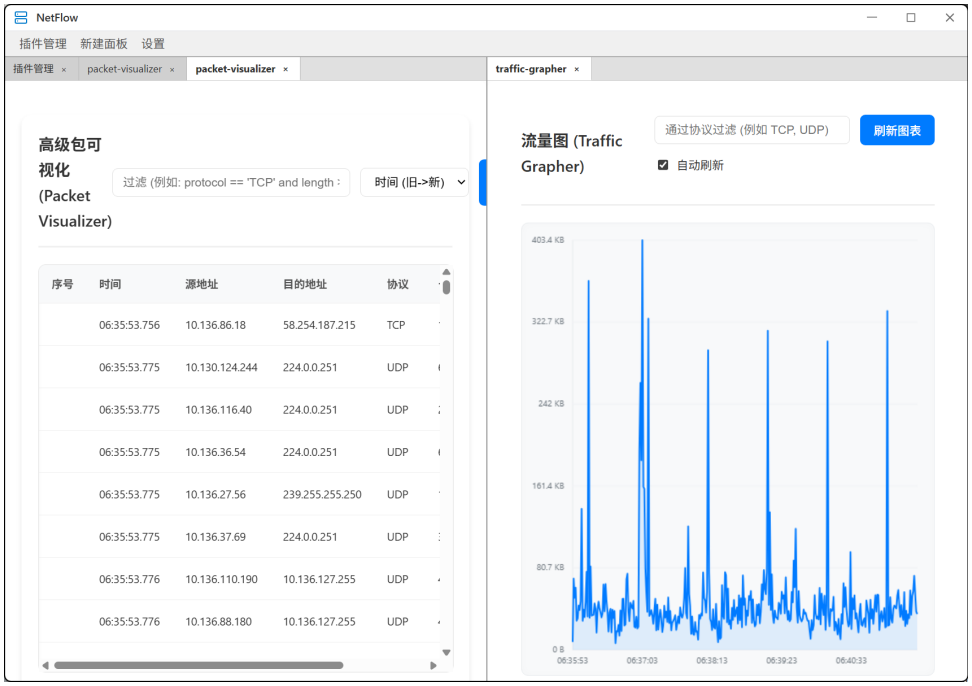

图 3: 应用主界面与 Docking 系统

6.2 抓包控制

图 4 展示了 Python Sniffer 抓包插件的运行状态。界面中包含了开始/停止控制按钮以及过滤器输入框，用户可以在此输入类 BPF 风格的过滤表达式（如 `src 192.168.1.1`）来筛选特定的网络流量，底层通过 Python 脚本实时捕获并转发数据。

6.3 数据包列表视图

图 5 展示了数据包可视化插件的主视图。该插件使用虚拟列表技术渲染大量数据包，保证了在处理数万条记录时的流畅度。列表清晰地展示了时间戳、源/目的 IP、协议类型和长度等关键信息，并支持点击查看详情。

6.4 深层数据分析

图 6 展示了单条数据包的详细信息视图。点击列表中的数据包后，会弹出此模态窗口。上方显示协议头解析出的关键字段，下方提供完整的十六进制（Hex）与 ASCII 对照视图，方便安全研究人员深入分析 Payload 内容。

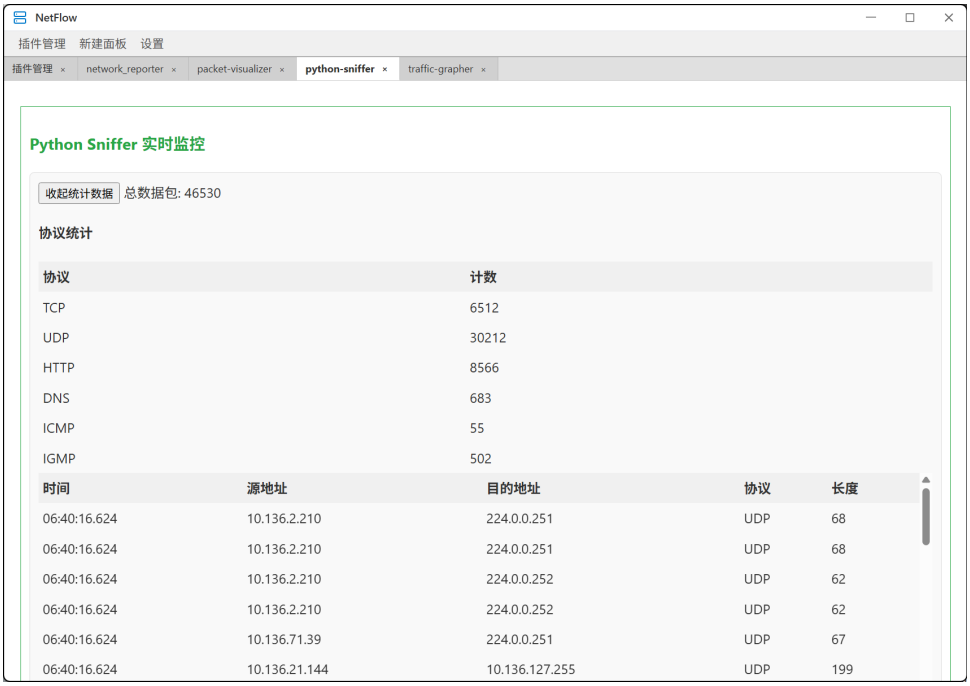

图 4: Python Sniffer 抓包运行中

6.5 流量趋势监控

图 7 展示了实时流量监控插件。该插件利用 HTML5 Canvas 绘制动态折线图，实时反映网络流量的吞吐量变化趋势。图表支持自动刷新，并能通过鼠标悬停查看具体时间点的流量统计数据，帮助用户快速识别流量峰值。

6.6 报告生成

图 8 展示了网络报告生成插件。用户可以在此配置报告标题、包含的数据包数量限制以及协议过滤器。点击生成后，系统将基于当前捕获的数据自动生成一份包含统计图表（如 Top Talkers、协议分布）和详细数据的 PDF 报告。

6.7 生成的 PDF 报告示例

图 9 展示了系统生成的最终 PDF 报告。报告内容排版整洁，包含了本次抓包会话的元数据（开始时间、持续时间、总包数）、关键统计指标以及自动生成的图表。该报告可直接用于网络审计或教学演示。

6.8 插件管理

图 10 展示了全局插件设置界面。用户可以在此管理已安装的插件，启用或禁用特定功能，并查看插件的版本和权限信息。这种模块化管理界面确保了系统的可扩展性和用户对功能的完全控制。

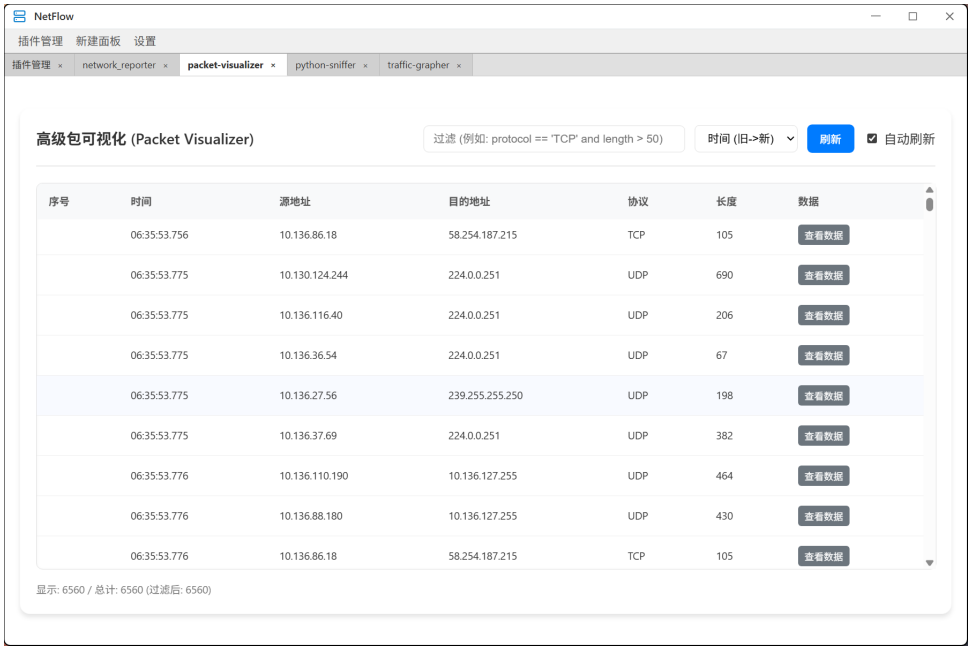

图 5: 数据包可视化列表 (Packet Visualizer)

7 程序亮点

- **模块化插件架构：**插件以声明式方式注册、沙箱化执行。主程序仅提供底座，所有业务功能均由插件实现，极大地提高了系统的可扩展性。
- **高性能渲染：**
  - **虚拟列表：**Visualizer 插件通过虚拟滚动轻松支撑 10w+ 数据包的流畅展示。
  - **Canvas 绘图：**Grapher 插件利用 Canvas 绘制实时波形，CPU 占用极低。
- **灵活的过滤引擎：**Sniffer 插件利用 Python 的动态特性实现了强大的过滤表达式求值，支持 `src`, `dst`, `proto` 等关键字及逻辑运算。
- **Docking 与 Popout：**支持多视图、多屏演示，用户可以将感兴趣的图表或列表弹出为独立窗口，适应多显示器工作环境。

8 致谢

感谢老师对课程的详细讲解，通过本次作业，我深入理解网络抓包与可视化技术，并锻炼了跨平台GUI应用开发与插件架构设计的能力。

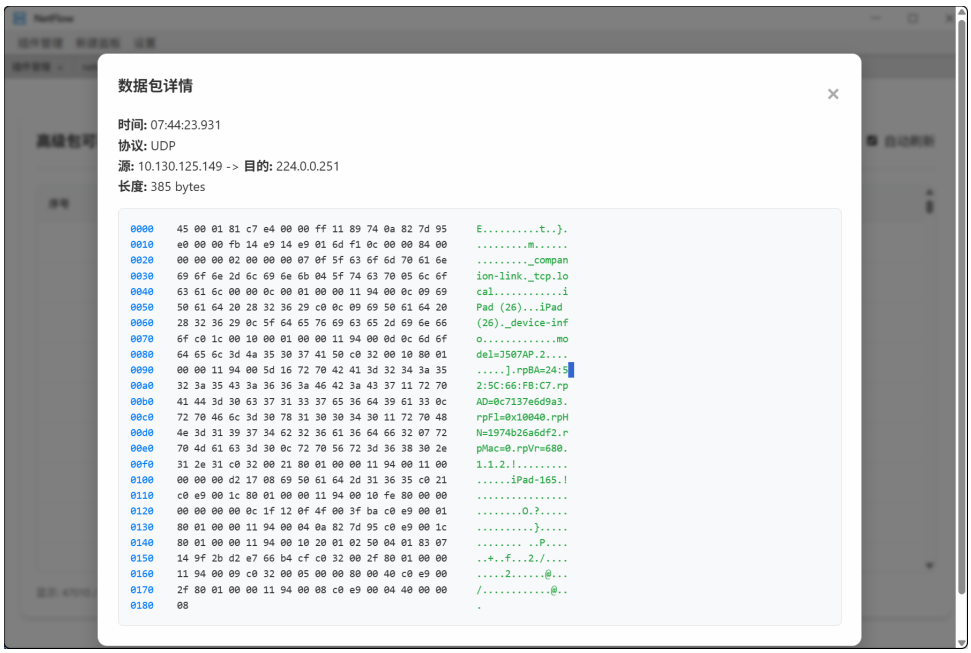

图 6: 数据包 Hex 详情视图

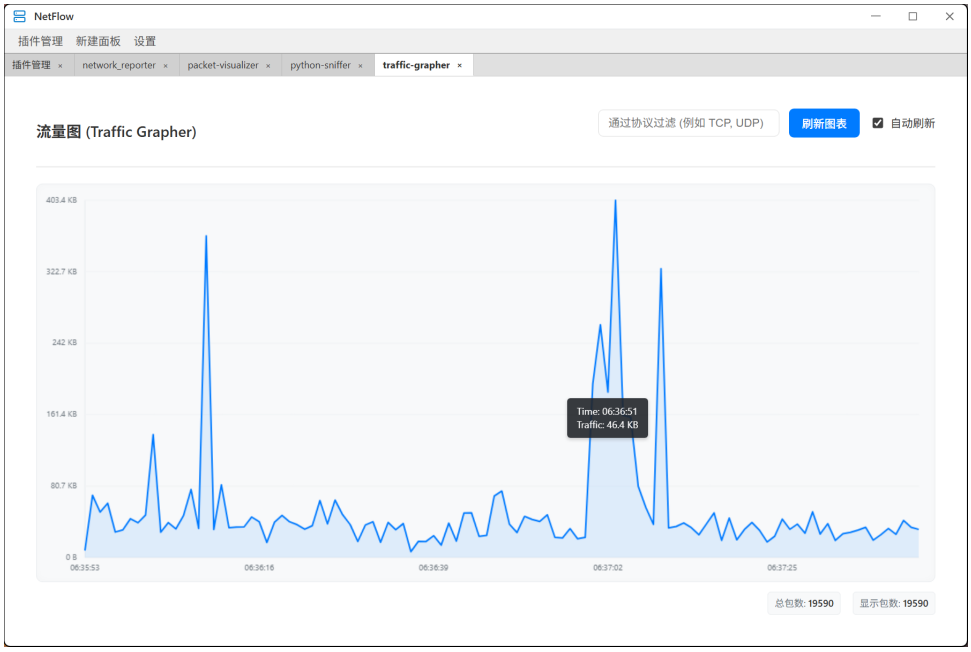

图 7: 实时流量图表 (Traffic Grapher)

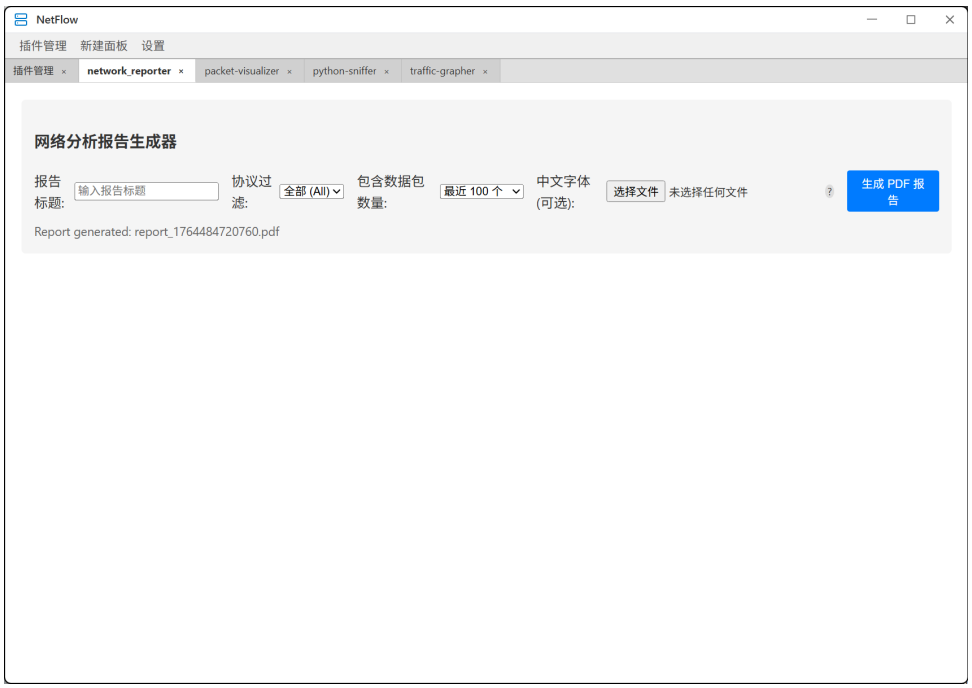

图 8: 网络报告生成器 (Network Reporter)

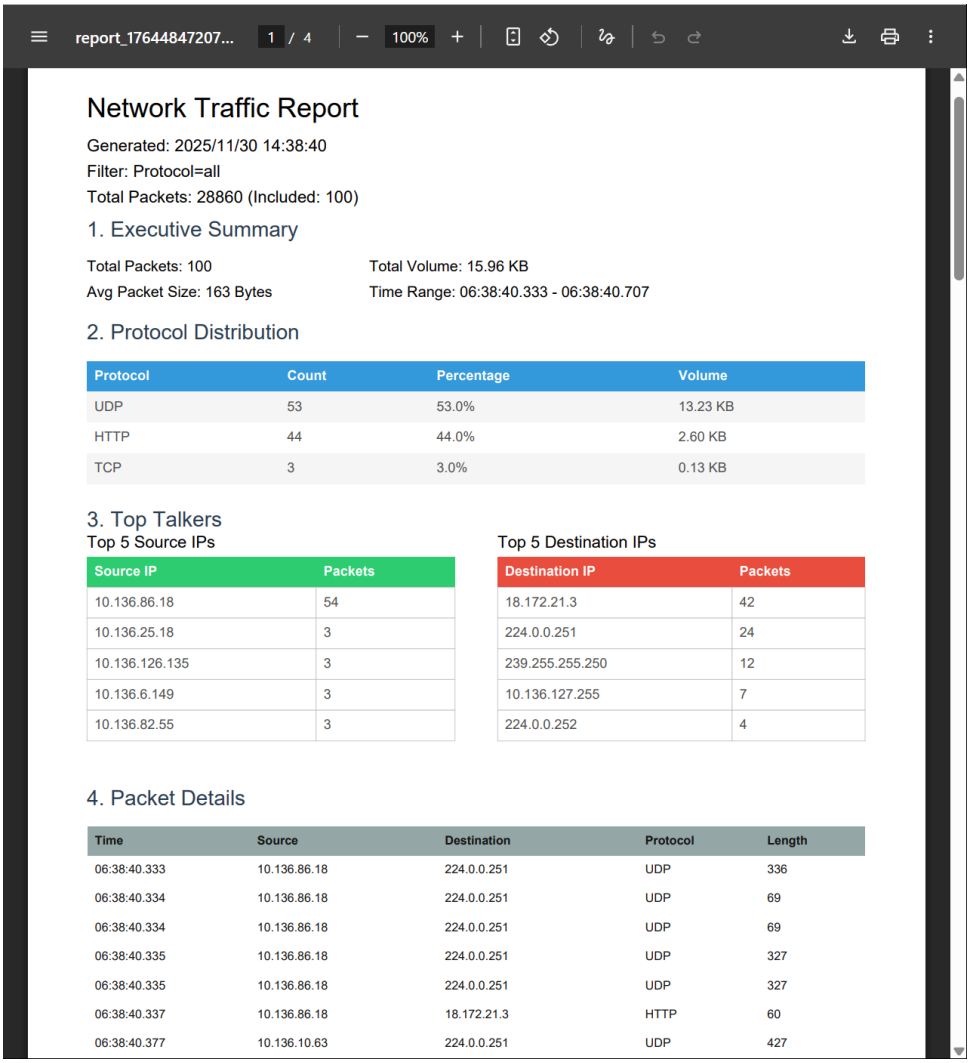

图 9: 生成的 PDF 流量报告预览

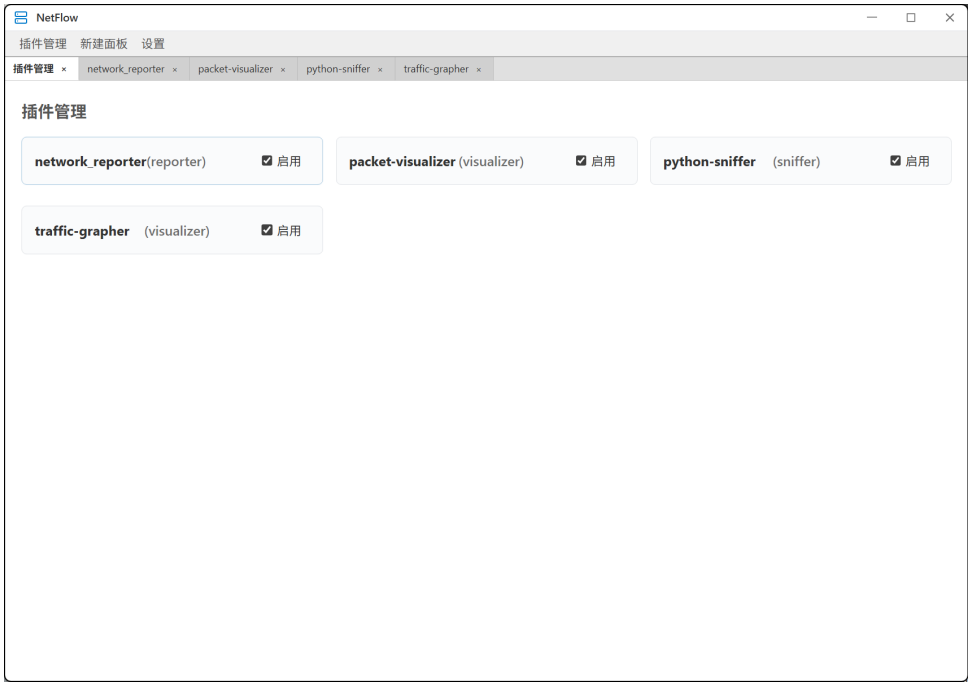
NetFlow
插件管理
新建面板
设置
插件管理
network_reporter
packet-visualizer
python-sniffer
traffic-grapher
插件管理
network_reporter(reporter)
启用
packet-visualizer (visualizer)
启用
python-sniffer (sniffer)
启用
traffic-grapher (visualizer)
启用

图 10: 插件设置界面