东 莞 理 工 学 院

本 科 毕 业 论 文

|  |  |  |
| --- | --- | --- |
| 毕业设计题目： | 网关 | |
| 学生姓名： | 周梓健 | |
| 学 号： | 202041417344 | |
| 院 系： | 计算机科学与技术学院 | |
| 专业班级： | 计算机科学与技术3 班 | |
| 指导教师姓名及职称： | | 任子良 副教授 |
| 起止时间： 2024年 12月--2025年 5月 | | |

**摘要**

随着云原生应用和微服务架构的发展，传统的单体应用逐渐被多个独立服务所替代，这种分布式系统架构虽然在灵活性和扩展性上有显著提升，但也带来了新的挑战，尤其是在流量管理、安全性、服务互操作性和系统监控等方面。为了有效应对这些挑战，网关作为中介层的解决方案应运而生，成为现代系统架构中的重要组成部分。在没有网关的架构中，每个服务需要独立处理请求路由、负载均衡、协议转换、身份验证、安全策略等多项功能，这不仅增加了服务的开发与运维成本，还导致了系统高度耦合，难以实现灵活的扩展和升级。此外，服务间缺乏统一的流量管理和监控，导致系统维护的复杂度大幅提高。现代网关能够通过集中管理流量、策略和安全认证，简化系统架构并提升系统可维护性和可扩展性。

本文的主要工作如下：

1. 对每连接一线程 、多路复用 、反应堆三个 I/O 架构进行性能测试，同时进行非阻塞优化，并选择 Reactor 作为最终实现网关的架构。
2. 实现 TCP、WebSocket、HTTP 协议的代理，包含对 TLS 连接与 HTTP 内容压缩的支持。
3. 实现流量的过滤、速率控制与负载均衡。
4. 实现 OAuth外部身份验证（微服务的访问控制），心跳检测（动态管理）、gRPC 管理接口（自动化配置）与 Docker（容器化，移植性）等云原生功能。

**关键词：**云原生网关，高性能服务器，操作系统

云原生（Cloud Native） 是一种架构理念，旨在利用云计算的 弹性、分布式特性和自动化能力，构建 高可扩展、易维护和高可靠 的应用。云原生架构微服务化、容器化、动态化、自动化等特性能高效地适应现代复杂业务需求。

**目录（这里页码是乱的）**

[第1章 引言 1](#_Toc164630450)

[1.1背景介绍 1](#_Toc164630451)

[1.2问题陈述 1](#_Toc164630452)

[1.3研究目标和意义 2](#_Toc164630453)

[1.4论文的主要工作 3](#_Toc164630454)

[1.5章节安排 3](#_Toc164630455)

[第2章 相关技术和理论模型 5](#_Toc164630456)

[2.1深度学习相关理论 5](#_Toc164630457)

[2.2姿态提取模块 12](#_Toc164630458)

[2.3相关开发环境 13](#_Toc164630459)

[2.4本章小结 15](#_Toc164630460)

[第3章 系统设计与实现 16](#_Toc164630461)

[3.1系统设计整体框架 16](#_Toc164630462)

[3.2 I/O 模型的对比与选择 17](#_Toc164630463)

[3.3 HTTP，Websocket，TCP 协议代理 17](#_Toc164630463)

[3.4 流量控制 18](#_Toc164630464)

[3.4.1 速率控制 20](#_Toc164630467)

[3.4.2 负载均衡 18](#_Toc164630466)

[3.4.3 流量过滤 20](#_Toc164630467)

[3.5 云原生功能支持 22](#_Toc164630468)

[3.5.1 OAuth 外部认证 22](#_Toc164630469)

[3.5.2 心跳检测 24](#_Toc164630470)

[3.5.2 gRPC 管理接口 24](#_Toc164630470)

[3.5.2 Docker 容器化 24](#_Toc164630470)

[3.6本章小结 30](#_Toc164630471)

[第4章 系统测试与评估 31](#_Toc164630472)

[4.1实验结果展示 31](#_Toc164630473)

[4.2模型准确度评估 33](#_Toc164630474)

[4.3系统存在的缺陷 38](#_Toc164630475)

[4.4本章小结 39](#_Toc164630476)

[第五章 总结与展望 40](#_Toc164630477)

[参考文献 41](#_Toc164630478)

[致谢 44](#_Toc164630479)

# 第3章 系统设计与实现

## 3.1 I/O 架构的测试与选择

## 3.2 I/O 架构的测试与选择

在服务器开发中，I/O（Input / Output，输入输出）是指程序与外部世界（如客户端、磁盘等）之间进行数据交换的过程。I/O 架构，则是指系统在处理这些 I/O 操作时所采用的组织方式和程序结构。它描述了程序如何发起 I/O 请求、如何等待 I/O 完成、以及在高并发场景下如何管理多个 I/O 连接之间的协作关系。

不同的 I/O 架构在处理模型、资源利用效率和并发能力方面各有特点。本小节将对常见的 I/O 架构进行性能测试，并选择最优者作为后续网关开发的基础架构。

3.2.1 每连接一线程（ Thread-per-Connection ）

在每连接一线程的架构中，对于每一个 TCP 连接，服务器都会启用一个新的线程处理后续的数据交换。但是在高并发的网络环境中，大量的线程创建将导致服务器资源快速枯竭，严重影响整体系统可用性。下面的示意图展示了这种架构的大致工作流程：

3.2.2 多路复用 （Multiplexing I/O）

不同于 3.2.1 中所述为每个连接都创建一个线程而导致大量资源消耗的架构，多路复用架构可以在单个线程上监听并处理多个连接。其原理是线程通过 select，poll 或epoll之一的系统调用向内核提交一组需要监听的文件描述符，并阻塞在该系统调用上。在某个文件描述符据准备就绪（连接状态建立或释放，有数据到达或出现文件结束符号等事件产生）的时候，系统调用返回，进程停止阻塞并可以执行相应的处理逻辑，之后进入下一轮事件循环。

在这样的架构下，一个线程能同时处理多个连接，有效地节省了系统开销。下面的示意图展示了这种架构的大致工作流程：

3.2.3 反应堆 （Reactor）

在上述 I/O 架构的处理流程中，网络数据的读写靠着 read ( ) 和 write ( ) 这样的系统调用 ( System Call )完成，默认情况下，这种读写将以阻塞的方式进行。

在操作系统中，阻塞（Blocking）是指进程在发起了一个系统调用之后，由于该系统调用的操作不能立即完成，需要等待一段时间，于是内核将进程挂起为等待状态， 以确保它不会被调度执行，占用 CPU 资源。而 Linux 操作系统将线程看作轻量级线程（LWP，Light-Weight Process）。所以阻塞的概念在线程调度中仍然适用。

使用阻塞的方式进行数据读写意味着在发起 read ( ) 和 write ( ) 这样的系统调用后，线程将等待至文件描述符就绪才会返回。假设程序以单线程的方式运行，在阻塞的期间，程序将无法接受新的连接，直到数据读写完毕。

在反应堆模式中，数据读写一般以非阻塞的方式进行。在 Unix 类的系统中，套接字可以通过设置 O\_NONBLOCK 标志位进入非阻塞模式。在这个模式下，假设在读写数据时数据并未准备就绪，线程将直接进入下一个事件的处理。

反应堆模式通常与事件队列相结合，每个工作线程都将不断检查队列中是否存在可以处理的事件，随后取出事件执行相应的处理函数。在这个架构下，工作线程的数量可以根据实际情况便捷地进行水平扩展。上述架构也被称为事件驱动架构，现代编程网络库中诸如Boost Asio， libevent 和 Netty 都采用了这样的事件驱动架构。

3.2.4 三种 I/O 架构性能测试

一个系统的吞吐量通常用 QPS 来进行描述，其计算公式为：

在不同的并发压力下，我们可以得到三种 I/O 架构的 QPS 如下图所示：

可见在高并发场景下，反应堆架构有着最好的吞吐量。后续我们将继续使用这个 I/O 架构完成网关的开发。

## 3.2 TCP，HTTP，Websocket 协议代理

3.2.1 总述

网关的基本功能是将到来的数据根据配置的目的地进行转发。但不同于传统的交换机在内网中简单地根据链路层地址进行转发，现代网关通常需要根据复杂的规则进行请求的处理，常见的例子包括：

1. 根据 HTTP 报文中的路由 （ url ）字段进行匹配，并在改写路由字段后发送数据
2. 在 HTTP 头部字段中寻找 Authorization 并提取 token 进行身份验证
3. 将 HTTP 请求转换为 Websocket 请求
4. 检查数据的 IP 地址和端口号是否在黑名单内并进行过滤

上述的例子说明网关应该具备一定的协议解析能力。现代操作系统编程接口通常提供 TCP 和 UDP 的数据解析能力，开发者只需要着重于应用层协议的处理。

本小节说明了网关对HTTP，Websocket 数据代理的过程，同时说明了 TLS 连接与 HTTP 内容压缩的实现。

3.2.2 HTTP 代理与内容压缩

3.2.3 Websocket 代理

3.2.4 TLS 连接