

东 莞 理 工 学 院

本 科 毕 业 论 文

毕业设计题目： 面向云原生架构的高性能流量网
关设计与实现

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

随着互联网架构的不断演进，系统间通信日益复杂，微服务架构逐渐取代传统单体架构成为主流。尽管当前已有如 Nginx 与 Envoy 等成熟网关方案在工业界得到广泛应用，但其配置复杂、使用门槛高、不支持静态资源代理或外部云原生认证等问题，使其在教学研究和轻量化部署场景下存在一定局限。为此，本文设计并实现了一款基于 C++ 的高性能自研网关系统 Azugate。系统支持 TCP、HTTP、WebSocket 协议代理及本地静态资源托管，集成了速率限制、轮询负载均衡、IP 哈希防火墙等服务治理模块。在云原生适配方面，系统支持 OAuth 认证、gRPC 远程配置、后端健康检查及 Docker 容器化部署。经性能测试，Azugate 可实现接近工业级网关 Nginx 92% 的请求处理能力，性能为 Golang 标准库 net/http 的 1.44 倍，Python SimpleHTTPServer 的约 30 倍。

本论文的主要工作如下：

- (1) 对每连接一线程、多路复用、事件驱动三种 I/O 架构进行性能测试，并选择事件驱动作为最终实现网关的架构。
- (2) 实现 TCP、WebSocket、HTTP 协议的代理，包含 TLS 连接、HTTP 内容压缩与零拷贝优化。
- (3) 实现基于 IP 的防火墙、速率控制与负载均衡。
- (4) 实现 OAuth 外部身份验证，心跳检测、gRPC 管理接口与 Docker 部署等云原生功能。

关键词：云原生网关，高性能服务器，操作系统

Abstract

With the continuous evolution of Internet architecture, inter-system communication has become increasingly complex. Microservice architecture is gradually replacing traditional monolithic systems as the mainstream approach. Although mature gateway solutions such as Nginx and Envoy are widely used in industry, their complex configuration, high usage threshold, and lack of support for static resource proxying or external cloud-native authentication introduce limitations in educational research and lightweight deployment scenarios. To address these issues, this thesis presents Azugate, a high-performance self-developed gateway system implemented in C++. Azugate supports TCP, HTTP, and WebSocket protocol proxying, as well as local static resource hosting. It integrates service governance modules including rate limiting, round-robin load balancing, and IP-hash-based firewall. In terms of cloud-native adaptation, the system provides OAuth authentication, gRPC-based remote configuration, backend health checking, and Docker-based container deployment. Performance evaluations demonstrate that Azugate achieves approximately 92% of the request handling throughput of the industrial-grade Nginx gateway, performs 1.44 times faster than Golang's standard net/http library, and is nearly 30 times faster than Python's SimpleHTTPServer.

The main contributions of this thesis are as follows:

- (1) Performance testing of three I/O architectures — one-thread-per-connection, multiplexing, and event-driven architecture. The event-driven architecture is ultimately selected as the architectural basis for Azugate.
- (2) Implementation of TCP, HTTP, and WebSocket proxies, including

support for TLS connections, HTTP compression, and zero-copy optimization.

(3) Implementation of IP-based firewalling, rate limiting, and load balancing mechanisms.

(4) Integration of cloud-native features including OAuth-based external authentication, heartbeat detection, gRPC management interface, and Docker deployment.

Keywords cloud-native gateway, high-performance server, operating system.

目录

第1章 引言	1
1.1 课题背景与研究意义	1
1.2 论文主要工作	2
1.3 章节安排	2
第2章 Azugate 网关系统架构设计与技术选型	4
2.1 Azugate 网关系统总体架构设计	4
2.2 相关开发环境与工具	5
2.2.1 相关开发环境与工具	5
2.2.2 相关开发环境与工具	5
2.3 本章小结	6
第3章 Azugate 网关系统设计与实现	7
3.1 I/O 架构的测试与选择	7
3.1.1 每连接一线程	7
3.1.2 多路复用	7
3.1.3 事件驱动	8
3.1.4 三种 I/O 架构性能测试	9
3.2 TCP HTTP WebSocket 协议代理	10
3.2.1 概述	10
3.2.2 路由匹配模块	10
3.2.3 TCP 与 TLS 连接	12
3.2.4 HTTP 代理与相关优化	13
3.2.5 WebSocket 代理	17
3.3 安全控制模块	18
3.3.1 概述	18
3.3.2 IP 防火墙	18
3.3.3 OAuth 外部认证	19
3.4 流量控制	21
3.4.1 速率控制	21

3.4.2 负载均衡	22
3.5 云原生功能支持	22
3.5.1 概述	22
3.5.2 gRPC 远程管理接口	22
3.5.3 心跳检测	23
3.5.4 Docker 容器化	23
3.6 本章小结	25
第4章 系统测试与评估	26
4.1 测试工具与环境	26
4.2 系统性能测试与功能验证	26
4.2.1 整体性能测试	26
4.2.2 核心功能验证	30
第5章 总结与展望	40
参考文献	41
致谢	43

第1章 引言

1.1 课题背景与研究意义

随着互联网架构的不断演进，系统间通信的复杂性日益提高。早期的 Web 系统往往采用单体结构，通信路径简单，服务器直接处理来自客户端的请求。然而，随着服务规模的扩大与系统职责的细化，微服务架构逐步成为主流选择。在这一架构下，系统被拆分为大量独立服务，服务间的通信频繁，且面临跨协议、跨网络、跨环境的种种挑战。为此，网关（Gateway）这一组件逐渐成为系统架构中不可或缺的关键角色^[1]。

网关的雏形可以追溯到反向代理服务器的广泛应用，例如 Nginx、Apache HTTPD 等，这些工具最初主要承担请求转发与负载均衡的功能。在微服务时代，传统的反向代理逐渐演化为具备更强服务治理能力的 API 网关^[2]，典型代表如 Kong、Traefik、Envoy 等。这些现代网关不仅支持多协议代理，还集成了认证授权、限流熔断、监控追踪等一系列功能，成为服务网络的统一入口，兼顾了安全性、可观测性与灵活性。

尽管诸如 Envoy、Nginx 等成熟网关方案已在工业界广泛应用^[3]，但在教学研究或轻量化部署等特定场景下，这些系统存在一定的局限性。例如，Envoy 缺乏对静态资源的直接代理能力，不适合承担前端资源托管任务；而 Nginx 则难以对接外部云原生认证系统^[4]，不利于现代服务网格下的统一身份控制。此外，它们通常伴随复杂的配置机制与较高的使用门槛，不利于开发者对网关底层原理的深入理解和灵活控制。

为应对教学研究、实验验证与轻量化生产部署等场景中对易用性、可控性及性能的综合需求，本文设计并实现了一款具备高性能通信能力与基础云原生特性的自研网关系统 Azugate。Azugate 以 C++ 语言为开发基础，采用事件驱动架构与异步非阻塞 I/O 模型，并通过自研解析器实现了高效的 HTTP 请求处理与多协议分发机制。在协议支持方面，系统兼容 TCP、HTTP 及 WebSocket 等常见协议，并提供基于 HTTP 的本地静态资源代理能力，适应多样化的服务接入需求。

在功能扩展方面，Azugate 集成了多项服务治理机制，包括基于令牌桶算法的速率限制、基于轮询策略的负载均衡、以及基于 IP 哈希匹配的访问控制模块），以提升系统在高并发场景下的稳定性与安全性。与此同时，系统还在底层实现中融入多项性能优化手段，如基于前缀树的高效路由匹配、状态机驱动的 HTTP 报文解析器、内容压缩支持、以及通过 Linux sendfile 系统调用实现的零拷贝传输。经

大量实验测试验证，Azugate 在吞吐能力、资源利用率及可扩展性方面均达到预期设计目标。

此外，为更好地契合现代云原生服务治理体系，Azugate 还引入了若干云原生特性：通过 OAuth 协议对接外部认证服务，借助 gRPC 实现统一的远程配置与管理接口，内置心跳检测机制用于后端服务的健康检查，并支持基于 Docker 的容器化部署，以提升系统在多环境下的可移植性与运维便捷性。

1.2 论文主要工作

本文主要围绕一款自研高性能网关系统 Azugate 的设计与实现展开，旨在在兼顾性能与易用性的前提下，为教学研究及轻量化部署场景提供一套低门槛、高扩展性的解决方案。首先，针对不同 I/O 架构的性能差异，本文设计并实现了每连接一线程、多路复用以及事件驱动三种模型的对比测试，最终选择事件驱动架构作为系统的实现基础。在此基础上，系统支持 TCP、HTTP、WebSocket 多协议代理，并实现了 TLS 安全通信、HTTP 内容压缩及基于 sendfile 的零拷贝优化机制。为提升服务的可控性与健壮性，Azugate 进一步集成了 IP 哈希防火墙、令牌桶速率限制、轮询负载均衡等服务治理模块，同时在云原生方向引入了 OAuth 外部认证、后端心跳检测、gRPC 管理接口与 Docker 容器化部署能力。通过对系统性能进行定量评估，结果表明 Azugate 能在保持较低资源消耗的前提下，实现接近 Nginx 的处理能力，并在多个典型场景下优于 Golang 与 Python 的标准服务器实现，验证了其在实际应用中的可行性与高效性。

1.3 章节安排

本论文的章节安排如下：

引言：本章介绍课题的研究背景、发展现状以及本项目的研究意义，指出当前高性能网关系统面临的挑战，并明确本文的研究目标和主要工作，为后续章节的展开提供理论基础和技术动因。

Azugate 网关系统架构设计与技术选型：本章对 Azugate 网关系统的总体架构进行设计说明，从整体功能出发，明确系统各模块之间的协作关系。同时介绍本系统的开发环境、工具链及其依赖的第三方库，为后续模块的实现提供开发基础和技术支撑。

Azugate 网关系统设计与实现：本章是全文的核心部分，详细阐述了 Azugate 网关各模块的设计与实现过程。首先通过对分析三种主流 I/O 架构（每连接一线程、多路复用、事件驱动），选定事件驱动模型作为系统底层框架。接着介绍了

TCP、HTTP、WebSocket 协议代理的实现及其相关优化技术，包括 TLS 加密通信、内容压缩与零拷贝传输。随后依次实现了 IP 防火墙与 OAuth 认证机制等安全控制模块，速率限制与负载均衡等流量控制机制，并在云原生方向引入 gRPC 管理接口、后端心跳检测与 Docker 容器化部署能力。最后对本章内容进行总结。

系统测试与评估：本章从实际部署出发，介绍了用于验证 Azugate 系统性能的测试工具与环境，随后通过系列测试验证了系统的整体吞吐性能、资源利用率及其在多种典型应用场景下的响应能力。同时对系统关键功能进行了逐项验证，证明其具备良好的稳定性与实用性。

总结与展望：本章总结了本文的主要研究工作与成果，归纳了 Azugate 系统在设计、实现与测试过程中所采用的关键技术与优化方法。同时对系统当前的局限性进行了分析，并对未来进一步提升系统性能、拓展功能、增强生态适配性等方向进行了展望。

第 2 章 Azugate 网关系统架构设计与技术选型

2.1 Azugate 网关系统总体架构设计

Azugate 网关系统整体架构由各模块以及负责不同功能的线程与一个任务队列组成，架构图如下所示：

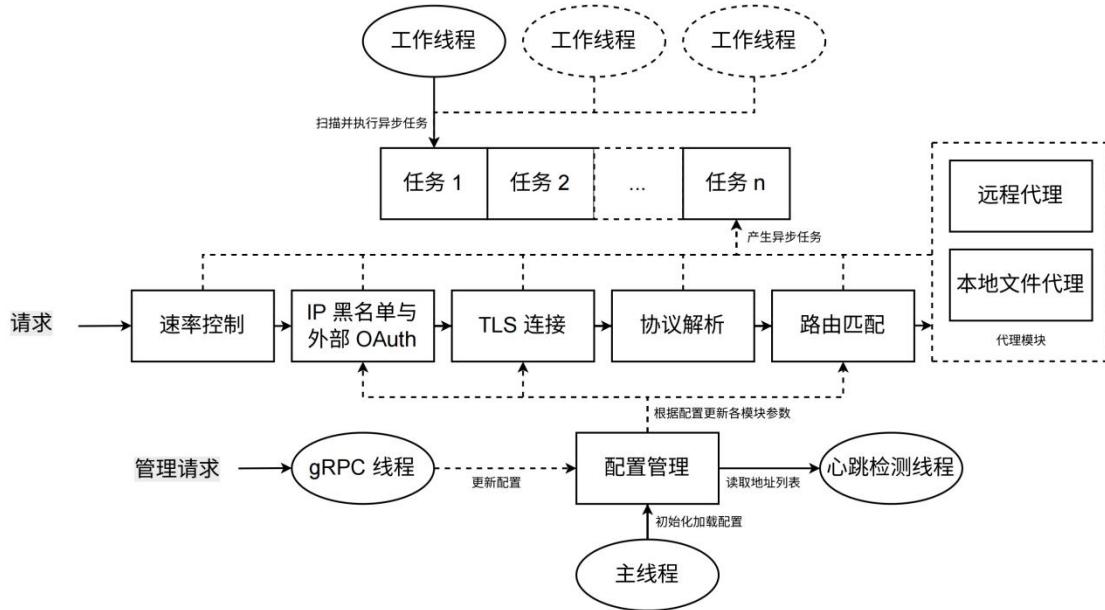


图 2-1 Azugate 系统架构图

每个普通客户端的请求在进入网关后，将依次经过以下处理模块：首先是流量过滤模块，用于拦截来自黑名单 IP 的连接，并进行 OAuth（Open Authorization，开放授权协议）身份认证；随后通过速率控制模块限制请求频率，以防止恶意请求造成系统过载；接着进入 TLS（Transport Layer Security，传输层安全协议）模块，负责建立安全连接；之后由协议解析模块对请求内容进行解析，支持 HTTP 与 WebSocket 协议；随后通过路由匹配模块，根据请求的路径与方法匹配路由，得到候选结果集合；在负载均衡模块中，通过轮询等策略在候选集合中选择目标后端；最后，Azugate 根据匹配结果决定将请求转发至远程服务器，或直接返回本地文件数据。

Azugate 网关系统主要由四个核心线程组成：主线程、gRPC（Google Remote Procedure Call，谷歌开源的高性能远程过程调用框架）管理线程、工作线程和心跳检测服务线程。其中，主线程负责读取配置文件、初始化 IO 框架，并启动 gRPC 管理线程和工作线程。gRPC 管理线程提供管理接口，支持外部通过 gRPC 协议发

送控制指令，实现对网关运行参数的动态修改，如更新速率控制上限、启用或关闭 TLS 支持、调整路由表等。工作线程则负责处理实际的客户端请求，通过轮询和执行任务队列中的非阻塞任务来实现高效并发处理。该设计支持工作线程的横向扩展，即通过增加线程数量可提升网关的并发处理能力，从而适应不同规模的网络负载。而心跳检测服务线程根据动态配置的地址列表，按照时间固定间隔检测服务是否健康运行。

2.2 相关开发环境与工具

2.2.1 相关开发环境与工具

表 2-1 开发环境与版本

类别	版本号
Operating system	Ubuntu 24.04
CPU	12th Gen Intel(R) Core(TM) i5-12600K
C++	17
CMake	3.25.0
vcpkg	8.9.2
Docker	27.3.1

2.2.2 相关开发环境与工具

(1) Boost.Asio: Boost.Asio 是一个跨平台的异步 IO 编程库，提供了统一的接口来处理底层系统的网络、定时器、信号等异步事件，支持同步和异步操作，广泛用于构建高性能网络服务器与客户端应用。该库设计基于事件驱动架构（Event-Driven Architecture），隐藏了底层平台差异，是构建分布式系统或高并发程序的重要组件。

(2) spdlog: spdlog 是一个高性能的 C++ 日志库，具有轻量级、接口简单、无依赖、开箱即用的特点。它支持异步日志写入，能够在高并发场景下极大减小日志对主流程的性能影响。spdlog 支持多种日志格式、文件滚动（按大小或日期分割）、控制台输出彩色日志等特性，适用于从嵌入式项目到大型服务器程序的各种应用。

(3) OpenSSL: OpenSSL 是当前最广泛使用的开源密码学库之一，提供了全

面的加密算法和协议支持。常用于实现 RSA、AES 等对称与非对称加密算法，MD5、SHA256 等哈希算法，以及 Base64 和 Base64Url 编码。OpenSSL 还实现了 TLS（传输层安全协议）协议栈，是构建 HTTPS、加密通信、证书管理等安全系统的核心组件。

(4) `nlohmann-json`: `nlohmann/json` 是一个现代 C++ JSON 库，以简洁易用而著称。它使用 STL 容器和操作符重载，使得 JSON 的构造、解析、访问、序列化与反序列化过程自然、直观。支持 JSON Schema、Unicode、兼容标准 JSON 文本格式，是处理配置文件、网络数据交换、API 开发中不可或缺的组件。

(5) `jwt-cpp`: `jwt-cpp` 是一个专注于 JSON Web Token (JWT) 处理的 C++ 库，用于生成、签名、验证和解析 JWT。它支持多种签名算法，如 HS256、RS256 等，并与 OpenSSL 等加密库兼容。该库可以方便地在认证授权流程中实现用户身份验证和权限控制，是构建安全 RESTful API 和单点登录系统的关键工具之一。

(6) gRPC 是由 Google 开发的高性能、开源远程过程调用 (Remote Procedure Call, RPC) 框架，基于 HTTP/2 协议传输、使用 Protocol Buffers 作为接口描述语言与数据序列化方式。gRPC 支持多语言互操作，具备流式传输、双向通信、连接复用等现代特性，适用于构建分布式系统、微服务架构中的服务间通信^[5]。

2.3 本章小结

本章围绕 Azugate 网关系统的整体架构设计与技术选型进行了系统性阐述。首先介绍了系统的核心处理流程与模块划分，详细说明了从请求进入网关到最终完成路由转发的完整路径，涵盖流量过滤、速率控制、TLS 加密、协议解析、路由匹配、负载均衡等关键功能模块。同时分析了系统的多线程设计，包括主线程、gRPC 管理线程、工作线程与心跳检测服务线程的职责分工及其在高并发处理、服务健康维护等方面的作用，体现出架构在灵活性与扩展性方面的优势。随后，本章还列出了系统开发过程中所使用的操作系统、编译工具链等基本环境信息，并详细介绍了各依赖库的功能与选型依据，如 Boost.Aso 提供异步 I/O 支持、spdlog 用于高性能日志记录、OpenSSL 实现加密通信、nlohmann-json 处理配置数据、jwt-cpp 支持 OAuth 授权、gRPC 实现远程管理通信等，为后续系统实现提供了坚实的技术基础。

第 3 章 Azugate 网关系统设计与实现

3.1 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^{[6][7]}。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。

3.1.1 每连接一线程

在每连接一线程（Thread Per Connection）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：

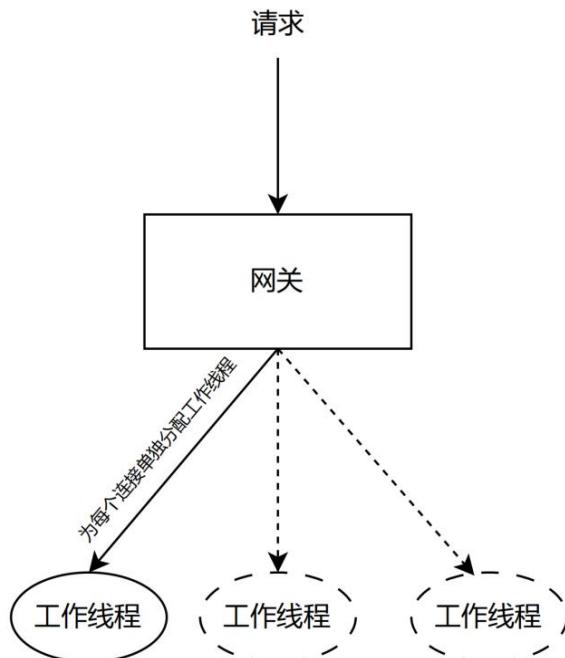


图 3-1 每连接一线程架构

3.1.2 多路复用

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（Multiplexing）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：

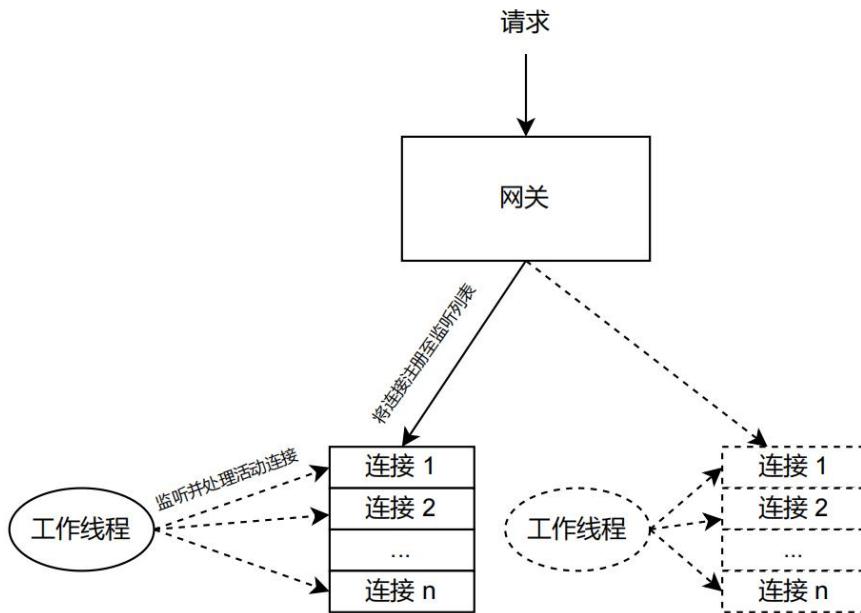


图 3-2 多路复用架构

3.1.3 事件驱动

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（Blocking）的方式进行。

在操作系统中，阻塞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^[8]。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。

在事件驱动架构中，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^[9]。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。下面的示意图展示了这种架构的工作流程，其中工作线程的数量同样可根

据需求横向扩展：

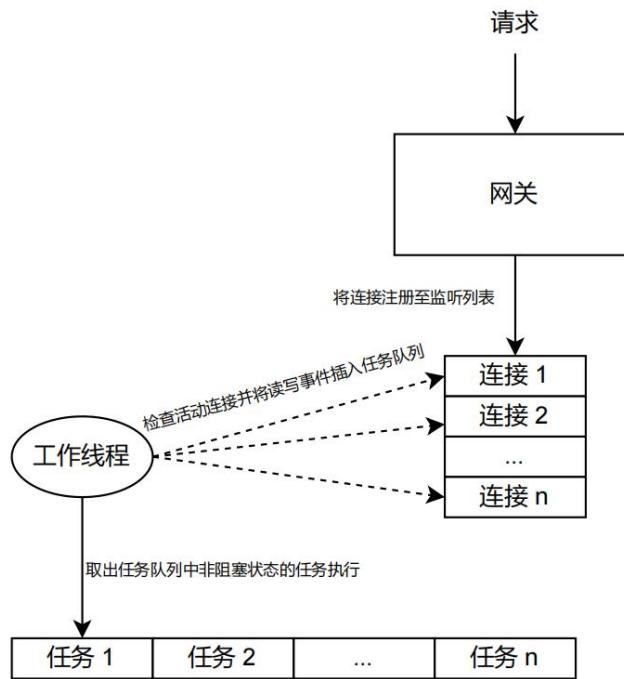


图 3-3 事件驱动架构

事件驱动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。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。

3.1.4 三种 I/O 架构性能测试

一个系统的吞吐量通常用 QPS (Queries Per Second) 来进行描述^[10]，其计算公式为：

$$QPS = \frac{\text{请求总数}}{\text{总耗时 (单位: 秒)}} \quad (3.1)$$

在 Intel Core i5-12600KF 芯片上，只使用单线程运行 I/O 架构。在接收到请求之后，为了模拟网络延迟，使当前线程睡眠 100 至 500 毫秒，并在接收到客户端请求后返回相同的页面。使用 wrk 压力测试工具模拟 10 分钟与 20 个 TCP 连接的压力，相同的实验重复进行 3 次并取平均数。我们可以得到三种 I/O 架构的性能如下表所示：

表 3-1 三种 I/O 架构性能对比

类别	版本号	平均响应时间 (ms)	最长响应时间 (ms)	QPS (s^{-1})
I/O 架构				
每连接一线程	1260.0	1820.0		3.26
多路复用	1180.0	1950		3.39
事件驱动	2.3	167.5		40234.2

由于每连接一线程与多路复用的 I/O 架构都会因为模拟的网络延迟阻塞当前线程，而事件驱动架构会在没有数据可读时检查队列中的下一个任务，所以事件驱动架构得到的测试结果都会远远优于前二者。可见在高并发场景下，事件驱动架构有着最好的吞吐量。Azugate 的开发即基于事件驱动的 I/O 架构。

3.2 TCP HTTP WebSocket 协议代理

3.2.1 概述

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：

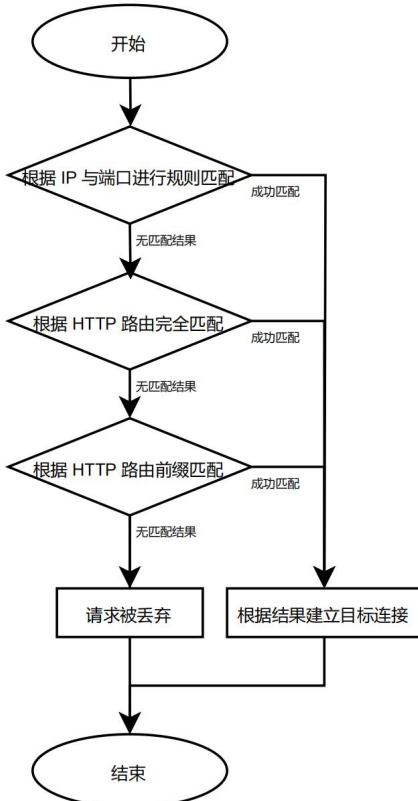
- (1) 根据 HTTP 报文中的请求路由进行匹配，并在改写请求路由字段后发送数据。
- (2) 在 HTTP 头部字段中寻找“Authorization”首部并提取令牌（Token）进行身份验证。
- (3) 将 HTTP 请求转换为 Websocket 请求。
- (4) 检查数据的 IP 地址和端口号是否在黑名单内并进行过滤。

上述的例子说明网关应该具备一定的协议解析能力。

本小节将详细介绍网关对 HTTP 与 Websocket 解析与代理的机制，并说明在传输安全（TLS 连接）与性能优化（HTTP 内容压缩与零拷贝）的关键实现细节。

3.2.2 路由匹配模块

当请求流量到达网关并成功通过流量过滤模块时，网关将会根据流量使用的协议与内部配置的规则决定流量应该向何处转发。本网关主要处理 TCP，HTTP 与 WebSocket 三种协议的匹配与转发。该模块工作流程如下图所示：



由于 TCP 属于 HTTP 与 WebSocket 的下层协议，配置的规则可能会发生冲突。因此本网关在不同网络层协议冲突时优先处理下层协议，即 TCP 协议将被优先处理，同时尽管 IP 地址的概念属于网络层而不是 TCP 协议本身，由于网络应用通常使用 IP 与端口号的组合标识网络地址^[10]，Azugate 在实现上提供了更加便利的支持。对于 HTTP 与 WebSocket 协议，匹配模式主要分为完全匹配与前缀匹配，当计算完全匹配未获得结果时会再次进行前缀匹配。

不管是 IP 与端口号组合或是 HTTP 路由匹配中的完全匹配，网关的实现都是通过在规则集合里遍历寻找值完全相同的字符串。而对于 HTTP 路由匹配中的前缀匹配而言，在规则集合中进行遍历将是低效的。假设多个规则都共享相同的前缀，这些前缀将会被重复遍历匹配。因此本网关的前缀匹配将借助前缀树（Trie）实现^[11]。前缀树通过共享相同前缀的节点来减少空间消耗，同时在查找时避免了多次遍历相同的前缀字符。用 n 代表平均字符串长度， m 代表存储的规则数量，查找的时间复杂度为：

$$O(n) \quad (3.2)$$

建立前缀树的时间复杂度为：

$$O(n \times m) \quad (3.3)$$

由于路由规则通常在网关运行前就规定好，所以建树时间可以忽略不计。在 Intel Core i5-12600KF 芯片上，假设平均规则长度为 15 个 ASCII 字符。不同数量的路由规则下，前缀树算法与暴力匹配算法的性能对比如下图所示：

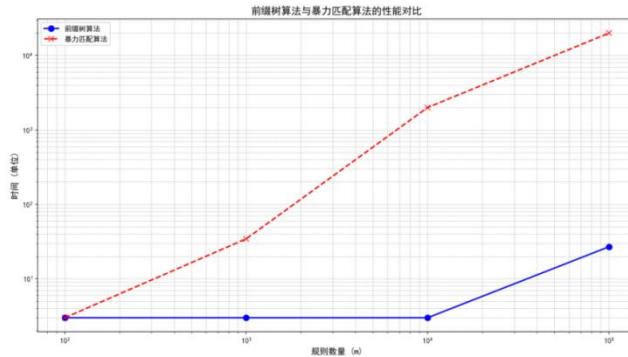


图 3-5 前缀树算法与暴力匹配算法性能对比

3.2.3 TCP 与 TLS 连接

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的传输层协议，具备可靠传输、顺序保证以及错误检测能力^[12]。它通过三次握手建立连接，四次挥手关闭连接，并在传输过程中依赖序列号、确认号与窗口机制保障数据的有序与完整^[10]。

现代操作系统已经对 TCP 协议进行了完整实现，因此在应用开发中通常无需手动处理 TCP 报文的拼装与重传，只需关注连接的建立、维持与释放过程。后文的 Websocket 与 HTTP 等应用层协议都建立在 TCP 的基础上，客户端的所有请求都将从建立 TCP 连接开始。TCP 连接与释放状态机如下图所示^[6]：

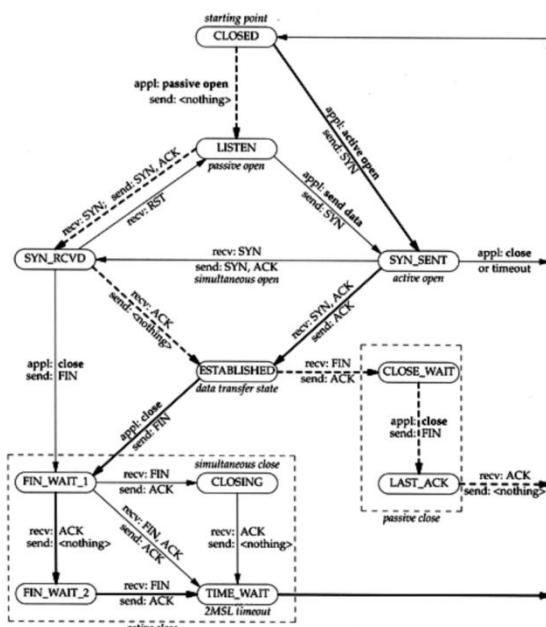


图 3-6 TCP 协议连接与释放状态机

在 TCP 提供可靠传输的基础上，实际部署中的网关通常还需对传输内容进行加密，以防止中间人攻击和敏感数据泄露。为此，TLS（Transport Layer Security）协议作为一种建立在 TCP 之上的加密层，被广泛应用于保护网络通信的安全。在 TLS 连接之上，应用层协议如 HTTP 和 WebSocket 可演化为加密版本，即 HTTPS 和 WSS（WebSocket over TLS）。

网关可以作为所有 TLS 流量的入口，在数据解密后将未受保护的数据转发至内网应用，也可以直接转发加密的数据至内网应用。前者称为透传模式(Pass-Through)，后者称为终止模式(Termination)^[13]。借助 OpenSSL 库，Azugate 实现了终止模式的数据代理。

3.2.4 HTTP 代理与相关优化

HTTP（HyperText Transfer Protocol，超文本传输协议）早期是一种基于 TCP 的应用层协议，广泛用于浏览器、客户端与服务器之间的数据交换。随着需求的发展，HTTP 协议经历了多个版本的演进，从传统的 HTTP/1.1^[14]，到支持多路复用与头部压缩的 HTTP/2^[15]，再到基于 UDP 和 QUIC 协议的 HTTP/3^[16]，逐步提升了性能与可靠性。尽管新版本在性能上有所改进，但当前大多数应用程序仍以 HTTP/1.1 为主，因其兼容性好、实现成熟，且部署更为广泛。HTTP/1.1 协议数据格式如下图所示：

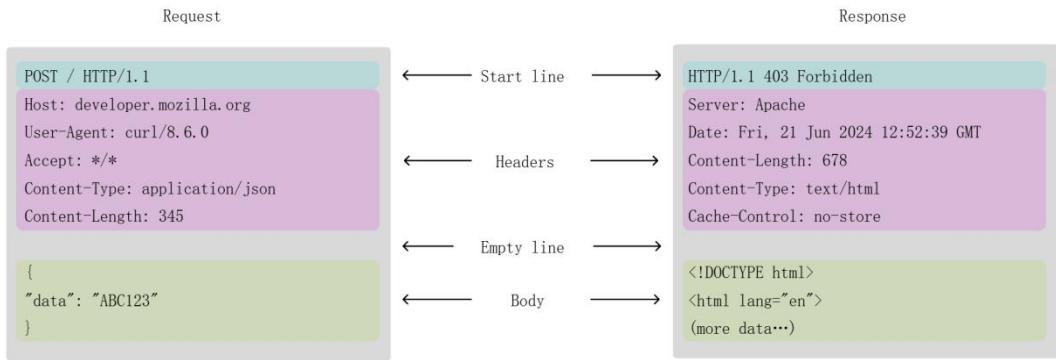


图 3-7 HTTP/1.1 报文格式

网络协议的解析通常由人为编写的状态机（State Machine）实现。状态机逐字符对到来的报文进行扫描，并及时更新内部状态。由于不涉及回溯与嵌套，状态机解析请求的时间复杂度为 $O(N)$ ，比使用正则表达式的字符匹配实现（复杂情况下能达到指数级的时间复杂度）更为高效。

Azugate 网关系统实现了对 HTTP/1.1 协议的解析，其中状态转移过程示意图如下：

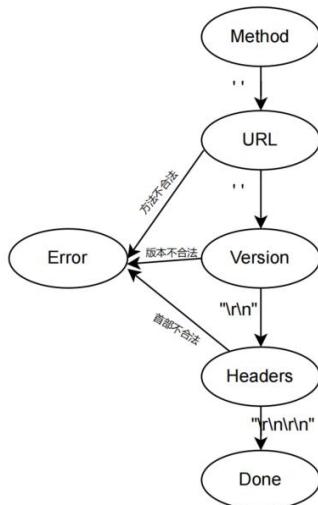


图 3-8 Azugate HTTP 报文解析器状态机

其中“Method”状态存在内部的状态转移，由于 HTTP 协议标准 RFC 2616 规定了请求方法（GET、POST、PUT、OPTIONS 等），“Method”状态内部应该在这些序列之间进行状态转移，其他的输入将会使状态机进入错误状态，提示调用者报文数据格式错误。同理于 HTTP 版本字段的解析，由于本论文网关只支持 1.1 版本的 HTTP 协议，在解析 HTTP 版本字段的时候应当逐字符解析到“HTTP/1.1”这个序列。部分 HTTP 请求方法状态转移示意图如下：

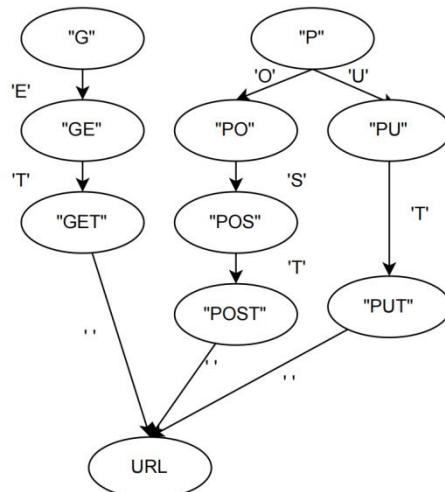


图 3-9 “Method” 状态内部

鉴于 HTTP 是应用最为广泛的网络协议，为了提升代理的性能与传输效率，Azugate 网关系统在 HTTP 代理模块中引入了零拷贝与（Zero-Copy）与 HTTP 内容压缩（Content-Encoding）两项优化手段：

(1) 零拷贝：

当进行 HTTP 代理时，若路由匹配模块计算后得到的目标路径为本地文件

(比如使用网关代理前端网页或静态文件)，即进行文件代理服务，则涉及服务器本地磁盘读写。在高并发环境下，本地磁盘读写通常会成为性能瓶颈。

传统文件传输通常需要服务器将文件数据从内核空间拷贝到用户空间，处理后再通过套接字将数据写回内核空间发送给网络设备。这种“多次拷贝”的过程会带来较大的 CPU 开销，尤其在进行大量静态资源转发时效率较低。为此本系统采用了零拷贝技术，通过内核态直接在文件系统与网络设备之间转移数据，避免不必要的用户态拷贝^[17]。二者对比如下图所示：

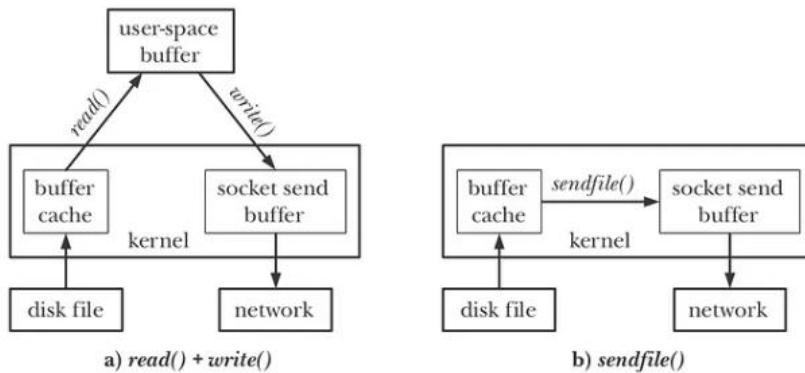


图 3-10 传统读写操作与零拷贝操作

在 Linux 中，常见的零拷贝机制包括 `sendfile()`, `splice()` 等系统调用，其中 `sendfile()` 系统调用适用于从文件系统向套接字传递数据。在 Intel Core i5-12600KF 芯片上，测试传统“`read() + write()`”读写与 `sendfile()` 对 HTML 网页进行代理的性能。其中测试程序 `wrk` 使用 4 线程与 100 个 TCP 连接，实验重复进行三次，每次进行 60 秒。结果如下表所示：

表 3-2 传统读写操作与零拷贝操作性能对比

文件传输方式	平均响应时间 (ms)	每秒处理请求个数 (/s)
<code>read() + write</code>	2.35	42029.64
<code>sendfile()</code>	1.37	72609.88

由表中数据得知使用 `sendfile()` 进行文件传输能大幅度提升文件传输性能。

(2) 内容压缩：内容压缩是一项常见的优化手段，用于减小传输数据的体积、降低带宽占用。标准 HTTP 协议中，与内容压缩有关的首部 (Header) 字段与常见值如下表所示，其中“*”表示接受任意压缩方式：

表 3-3 HTTP 内容压缩相关首部

首部名	常见值	说明
Accept-Encoding	gzip, deflate, br, *	客户端首次请求网页时，服务器将接受的压缩算法填入这个头部字段返回
Content-Encoding	gzip, deflate, br	用于指示数据载荷的压缩算法
Transfer-Encoding	chunked	设置为 chunked 的时候 Content-Length 字段无效

尽管对数据进行压缩能减轻网络压力，提升网络传输速度。但对于大文件的传输而言，数据的压缩与解压会消耗大量时间，引入不可忽略的网络时延。因此 gzip 与 br 等压缩库都提供了流式编程接口（Streaming API）以对数据分块压缩，以便应用程序在压缩数据的同时进行网络传输。

当“Transfer-Encoding”首部值设置为“chunked”时，HTTP 标准对数据载荷的格式做出了如下要求^[18]：

(1) 每个载荷应该包括压缩数据块的长度与压缩数据块，以两个 CRLF 代表载荷的结束。其中数据块的长度应该用十六进制进行表示，并将数值转换为 ASCII 字符串。压缩数据块长度与压缩数据块本身以 CRLF 符号分隔。

(2) 在所有压缩数据块都发送完毕之后，发送端应该发送一个长度为 0 的数据压缩块指示数据的结束。

假设一个文件需要分成两个压缩块进行发送，用 “[DATA]” 代表压缩数据块，一个内容压缩与分块传输的例子如下：

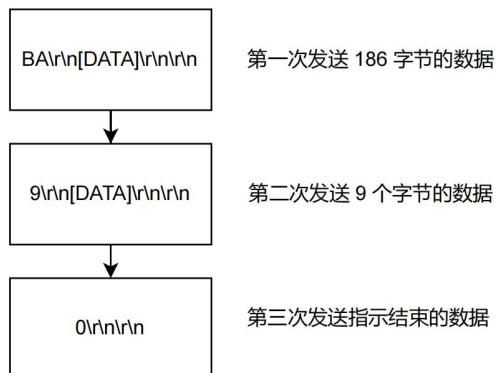


图 3-11 分块传输示例

Azugate 网关系统实现了基于 gzip 压缩算法的分块传输的支持，整体的请求与响应流程示意如下：

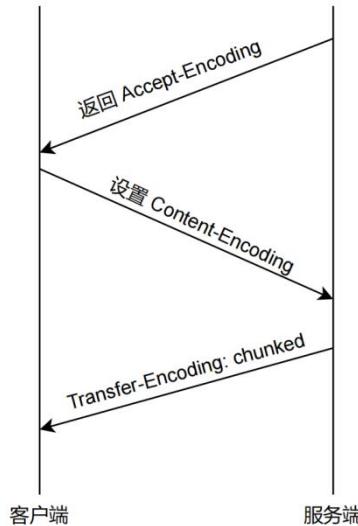


图 3-12 Azugate 内容压缩与分块传输功能工作流程

3.2.5 WebSocket 代理

WebSocket 是由 HTTP 协议扩展的全双工应用层协议^[19]，广泛应用于在线聊天与多人游戏等需要实时数据传输的场景。客户端为了将 HTTP 协议升级为 WebSocket 协议，需要向服务端发起 WebSocket 握手请求（一个方法为“GET”并设置了特别首部的 HTTP/1.1 报文）。客户端发起握手请求时需要设置的首部如下表所示：

表 3-4 WebSocket 客户端握手请求首部字段

首部名称	说明
Upgrade	通常值为 websocket
Connection	通常值为 Upgrade
Sec-WebSocket-Key	由客户端生成的随机字节

如果服务器支持 WebSocket 协议并同意协议升级，会将客户端请求中“Sec-WebSocket-Key”首部值与服务端生成的随机密钥拼接，将拼接字符串进行 SHA-1 哈希运算与 Base64 编码计算后得到“Sec-WebSocket-Accept”首部值并设置，同时返回状态码为 101（Switching Protocols）的响应报文，并设置下表中所示字段：

表 3-5 WebSocket 服务端握手响应首部字段

首部名称	说明
Upgrade	通常值为 websocket

Connection	通常值为 Upgrade
Sec-WebSocket-Accept	由“Sec-WebSocket-Key”生成

与上文所述代理 HTTP 协议相比，WebSocket 的数据帧的长度应从“Payload len”与 Extended payload length 字段中获取，同时应该为使用 WebSocket 的流量保持 TCP 长连接。WebSocket 协议数据帧格式如下图所示：

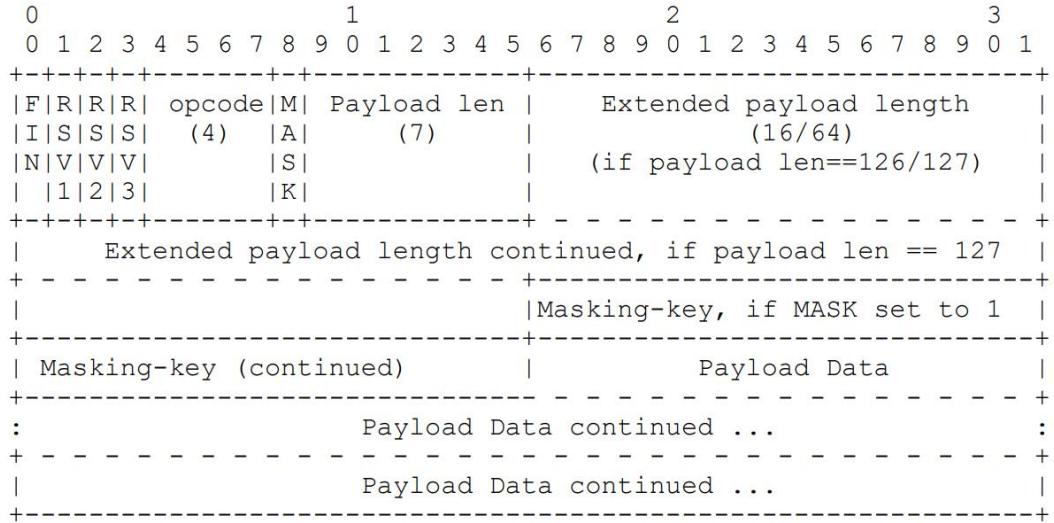


图 3-13 WebSocket 数据帧格式

由于 WebSocket 的握手首部与 HTTP/1.1 兼容，在 Azugate 网关系统的 WebSocket 代理实现中，将使用 3.2.4 节中的 HTTP 解析器同时对 WebSocket 与 HTTP 报文进行解析。当解析器在首部发现“Connection: Upgrade”这样的信息时，Azugate 将使用 WebSocket 的方式进行通信。

3.3 安全控制模块

3.3.1 概述

在现代微服务架构中，安全性是系统设计的关键考虑因素之一。Azugate 安全控制模块主要包括基于 IP 地址的防火墙机制、OAuth 外部认证与访问控制策略。其中，OAuth 协议虽广泛应用于云原生架构，但本系统中更强调其作为访问控制组件的角色，因此将其归入安全控制模块进行统一阐述。通过将这些关键功能整合至网关层，可有效减少服务之间冗余的安全实现逻辑，提升系统的统一性与安全性。

3.3.2 IP 防火墙

Azugate 网关系统在网络层实现了基于源 IP 地址的防火墙机制。当接收到

的请求源 IP 地址被列入黑名单时，网关将阻止其 TCP 连接的建立，从而实现对恶意流量的屏蔽。

为此，Azugate 内部维护了一个 IP 地址的哈希表。在编程层面，使用 C++ 标准库提供的 `std::unordered_set` 来存储和管理被封禁的 IP 地址^[20]。该容器在底层通过哈希函数将每个 4 字节的源 IP 地址映射到固定大小的桶数组中，并通过开放寻址或链表等方式处理哈希冲突。IP 地址被存储在一个 `uint32_t` 类型的变量中。设插入的新值为 x ，哈希表的长度为 n 。C++ 标准库对于这个类型的变量采用的哈希函数 $H(x)$ 如下：

$$H(x) = x \bmod n \quad (3.4)$$

3.3.3 OAuth 外部认证

在传统架构中，服务认证往往是由各个业务服务独立实现的，导致认证逻辑分散、安全策略不一致，难以维护和审计。但在微服务体系中，服务数量众多，若每个服务都自行处理认证，将会导致用户身份信息在多个服务主机上冗余存储。这种设计不仅增加了开发成本与维护成本，也有着很大的安全漏洞。

为了解决这一问题，现代生产环境往往会引入外部认证服务器统一存储并管理用户的身份信息。访问应用程序的请求应该先从外部认证服务器中获取身份令牌，之后携带身份令牌访问应用程序。由于在微服务架构中服务数量众多，且服务实例动态改变，若每一个服务都需要在外部认证服务器中先进行注册，将导致难以维护与难以扩展的问题。因此现代微服务集群中通常使用网关统一对接认证逻辑。

OAuth（Open Authorization）是一个用于资源授权的开放标准协议，最初由 Twitter 等公司提出，目前已广泛应用于 Web 与移动应用程序的第三方访问控制场景中^[21]。OAuth 2.0 协议的核心目标在于，使客户端应用程序在不直接接触用户凭据的前提下，通过令牌机制访问用户受保护的资源，从而提升系统安全性与用户隐私保护能力。

Azugate 网关系统代理 HTTP 协议的时候支持通过 OAuth 协议与外部认证服务器集成，统一对接认证逻辑。具体流程如下：

- (1) 用户访问应用程序前会被网关重定向至外部认证服务器获取 Code。
- (2) 用户向网关提交 Code，网关向外部服务器验证 Code 并获取用户身份信息。
- (3) 若网关成功验证用户身份信息，将返回访问应用程序的需要的访问令牌（Access Token）。

集成外部 OAuth 的网关认证流程示意图如下：

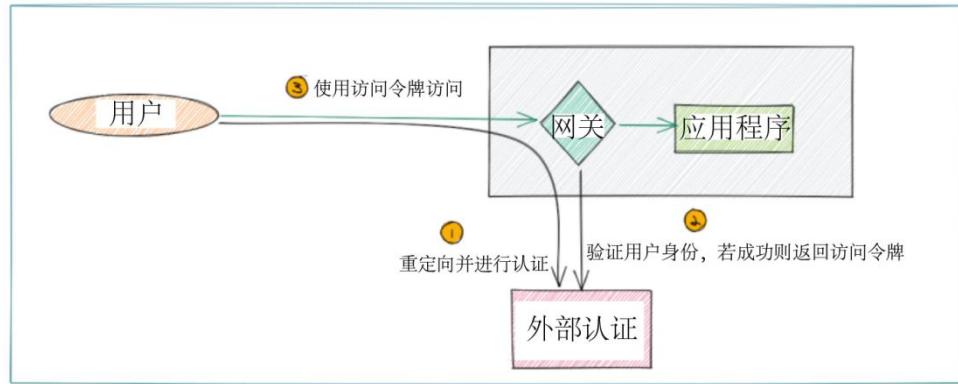


图 3-14 集成外部 OAuth 的网关认证流程示意图

Azugate 网关系统生成的访问令牌采用 JWT (JSON Web Token) 格式，这是一种基于 JSON 的轻量级令牌格式，广泛用于网络应用中安全传递身份验证信息。JWT 通常由三部分组成：头部 (Header)、载荷 (Payload) 和签名 (Signature)，签名部分可通过 HMAC (基于密钥的哈希消息认证码) 或 RSA (非对称加密算法) 等算法生成，以确保令牌的完整性与可信度。

在实际应用中，常用的 JWT 签名算法包括 HS256 (基于 HMAC 的 SHA-256) 和 RS256 (基于 RSA 的 SHA-256) 客户端可在后续请求中携带该令牌，用于传递用户身份信息，从而实现无状态的认证机制，避免低效的重复登录与验证过程。

JWT 的数据结构如图所示：

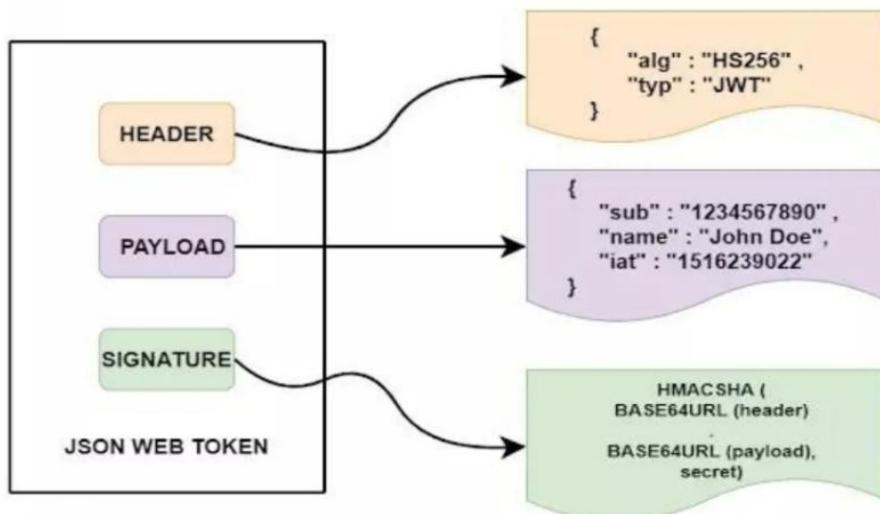


图 3-15 JWT 数据格式

上文所述的这种集中式认证机制不仅增强了系统的安全性和一致性，也简化了下游服务的开发负担，使得服务本身可以专注于业务逻辑，而无需重复实现认证功能。

3.4 流量控制

3.4.1 速率控制

速率控制的主要意义在于平衡系统负载，防止因短时间内大量请求的涌入导致系统过载或性能下降。它通过控制请求的处理速率，确保系统在高并发情况下仍能稳定运行。常见的速率控制算法包括令牌桶算法（Token Bucket），漏桶算法（Leaky Bucket）以及滑动窗口（Sliding Window）算法^{[22][23]}。三种算法的大致流程如下：

- (1) 令牌桶算法：令牌以固定速率放入桶中，请求到达时消耗令牌，若桶内有令牌则允许请求通过，否则拒绝或延迟。
- (2) 漏桶算法：请求按固定速率流入桶中，桶以恒定速率流出，若桶满则丢弃请求，确保输出流量平稳。
- (3) 滑动窗口算法：每个请求记录时间，检查当前时间窗口内的请求数量，若未超出限制则允许请求，通过时间滑动更新窗口。

对上述算法流程进行对比可发现漏桶算法和滑动窗口在遭遇突发的大规模流量时，会将超过容量部分的请求直接丢弃，这样的设计缺乏泛用性，难以应用在网关中。同时滑动窗口算法中的窗口大小较难调整：过小的窗口将导致频繁的请求数检查，占用 CPU 资源。而过大的窗口可能导致流量集中在窗口初始化初期，而窗口剩余的时间流量无法通过。

Azugate 网关系统采用了令牌桶算法作为速率控制的实现，由于令牌是按照均匀速率生成，相比起使用滑动窗口算法将获得更平滑的流量。而相比起漏桶算法直接丢弃超过桶容量的请求，使用令牌桶算法的服务器可以配置多余的请求等待新令牌生成后放行或直接被丢弃。令牌桶算法工作流程如下图所示：

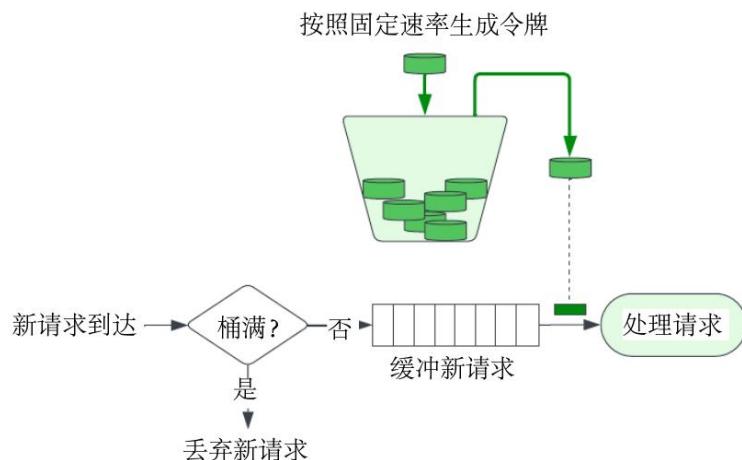


图 3-16 令牌桶算法工作流程

3.4.2 负载均衡

负载均衡在云原生和云计算系统中的意义在于通过主动配置将流量合理分配到不同的服务实例，确保系统的高可用性和稳定性。通过配置合理的路由策略，负载均衡能够优化资源利用率，避免单点故障，提高系统性能。

Azugate 网关系统负载均衡模块建立在 3.2.2 节所述的路由匹配模块之上，该模块计算得出一个目标主机的集合，并在这个集合中进行轮询并取出一个目标地址，随后处理流程进入代理模块。通过不同目标地址的分配实现了基本的负载均衡功能。

3.5 云原生功能支持

3.5.1 概述

随着云计算和微服务架构的发展，现代网关系统不仅需要具备基本的流量转发和路由能力，还应支持一系列云原生特性，以满足在弹性伸缩、服务治理、安全认证及自动化运维等方面的需求。本节将介绍 Azugate 网关系统在云原生场景下的功能支持情况，包括用于服务健康检测的心跳机制，便于远程配置和管理的 gRPC 接口，以及基于 Docker 的容器化部署方式。这些能力共同构成了网关在云环境下稳定运行、易于集成与扩展的技术基础。

3.5.2 gRPC 远程管理接口

在现代分布式系统中，管理和监控功能对确保应用的高效运行至关重要，特别是在微服务架构和网关的场景下。系统的管理接口提供了一种统一的方式，能够实时监控服务的健康状况、性能数据以及配置状态等关键信息。

gRPC（Google Remote Procedure Call）是一种高性能且开源的远程过程调用（RPC）框架^[5]，被广泛应用于 Google 云计算平台的基础设施中^[24]。它基于 HTTP/2 协议，并使用 Protobuf（Protocol Buffers）作为接口定义语言。与传统的 RESTful API 相比，gRPC 在传输过程中提供了更低的延迟和更高的吞吐量。通过二进制格式的数据传输，gRPC 避免了文本格式（如 JSON）带来的性能开销，从而使网络通信更加高效。此外，gRPC 生态支持多种编程语言，开发者无需过多关注底层网络细节，可以专注于应用逻辑的实现，从而实现跨平台、跨语言的高效网络通信。

Azugate 网关系统的各项配置参数不仅可以通过本地文件进行配置，还能够通过 gRPC 接口实现远程配置。通过 gRPC 生态系统中的服务反射（Service Reflection）功能，调用网关管理接口的程序可以自动快速地获取接口相关信息。与传统的互联网开发中需要事先编写复杂 API 说明文档的方式相比，这种设计显著提升了自动化开发效率，同时大大降低了后续接口维护的成本。本网关提

供的 gRPC 管理接口如下表所示：

表 3-6 Azugate 支持的 gRPC 服务接口

接口名称	说明
api.v1.GetConfig	获取当前配置相关信息
api.v1.UpdateConfig	对基本配置（例如开启 HTTP 压缩，TLS 连接，速率控制）进行更新
api.v1.GetIPBlackList	获取 IP 黑名单
api.v1.UpdateIPBlackList	设置 IP 黑名单
api.v1.GetRouter	获取路由设置
api.v1.UpdateRouter	更新路由设置
api.v1.RegisterHealthz	注册心跳检测服务

3.5.3 心跳检测

在分布式系统中，心跳检测是确保各个组件健康、服务可用性的关键机制之一^{[25][26]}。随着微服务架构和网关的普及，服务之间需要频繁地进行通信，确保互相之间的健康状态可以被及时检测到。如果某个服务因为故障或异常中断，心跳检测能够帮助网关及时发现并切换到其他健康的服务，从而提高系统的可靠性和可用性。

现代应用中的心跳检测通常有两种实现方式：一种是基于 TCP 连接的保活机制，另一种是基于应用层协议（如 HTTP 或 WebSocket）自行实现的心跳检测机制。基于 TCP 保活机制的检测方式不需要额外的系统模块，但它仅能保证 TCP 连接的存活，并不能完全反映应用程序的健康状态。而基于应用层协议的心跳检测需要应用程序统一实现心跳检测接口，以供网关进行定期检查。

Azugate 网关系统的心跳检测实现将基于应用层协议，并支持 HTTP 与 gRPC 协议。服务端需要根据预定义的 HTTP 接口或 gRPC 接口实现心跳检测功能，并通过网关配置文件进行注册。在生产环境中，A zugate 定期向服务端发送心跳请求，以检测其健康状态。如果服务端未能在规定的时间内响应心跳请求，A zugate 将视该服务为不可用，并采取相应的故障处理措施。

3.5.4 Docker 容器化

在现代分布式系统和微服务架构中，服务的快速部署、跨平台迁移以及环

境一致性是应用运行和维护的核心挑战^[27]。传统的部署方式通常依赖手动配置和环境管理，导致开发、测试、生产环境中常常出现配置差异和兼容性问题，从而影响服务的稳定性和可维护性。尤其在多云或混合云环境中，如何确保应用能够一致地在不同环境中运行，成为系统设计中的一大难题^[28]。

为了解决这些问题，Docker 容器化技术提供了高效且便捷的解决方案。通过将网关及其依赖打包到独立的容器中，Docker 确保了应用程序的环境一致性。开发者无需关心底层环境的配置差异^[29]。

为了支持 Docker 工具构建应用程序的运行环境，需要编写 Dockerfile。Azugate 网关系统的构建与部署将使用 Dockerfile 的多阶段构建功能：一个容器用于编译网关的二进制文件，另一个容器则用于运行已编译的二进制文件。

Docker 后台将根据此文件先进行程序构建工具与依赖第三方库的下载，之后进行代码编译，最后将二进制文件拷贝到最终分发的目标容器中。其中本论文网关使用的 Dockerfile 代码如下：

```
# 在构建容器中编译程序

FROM ubuntu:latest AS builder
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y build-essential cmake git curl pkg-config
WORKDIR /opt
RUN git clone https://github.com/microsoft/vcpkg.git \
    && ./vcpkg/bootstrap-vcpkg.sh
ENV VCPKG_ROOT=/opt/vcpkg
ENV PATH="${VCPKG_ROOT}:$PATH"
WORKDIR /project
COPY ..
RUN mkdir build && cd build && cmake --preset=default \
-DCMAKE_BUILD_TYPE=Release -DVCPKG_BUILD_TYPE=release ..
RUN cd build && cmake --build .
# 将二进制程序拷贝到目标容器

RUN apt-get update && apt-get install -y libstdc++6 && rm -rf /var/lib/apt/lists/*
COPY --from=builder /project/build/azugate /app/bin/azugate
COPY --from=builder /project/resources /app/resources
WORKDIR /app/bin
ENTRYPOINT ["./azugate"]
```

3.6 本章小结

本章节讨论了三种常见 I/O 架构（每请求一连接，多路复用，事件驱动）的优劣并进行性能测试对比，最终选择事件驱动模式作为网关的最终架构。接着讨论了网关对于常见协议（TCP，HTTP，WebSocket）的解析与代理，其中包括了路由匹配模块的设计、TLS 连接、HTTP 内容压缩优化与 Linux 系统下零拷贝调用优化。之后针对安全控制模块，讨论了 IP 防火墙以及 OAuth 外部认证功能。接着从速率控制与负载均衡三个方面讨论了网关对流量控制的实现。这些内容支撑起了网关的基本架构。

最后针对云原生相关功能，说明了 Azugate 网关系统对 gRPC 远程管理接口，心跳检测与 Docker 容器化部署等高级功能的实现。

第4章 系统测试与评估

4.1 测试工具与环境

本章节使用测试工具如下表所示：

表 4-1 系统测试工具

类别	版本号
Operating system	Ubuntu 24.04
CPU	12th Gen Intel(R) Core(TM) i5-12600K
wrk	2.1.1
Docker	27.3.1
Google Chrome	135.0.7049.85
Python	3.13
Golang	1.22.1
Nginx	1.27.4
Postman	11.40.6
Auth0	-
curl	8.7.1
Bytebase	3.5.2

为了构建一个隔离的网络环境，并更有效地限制被测试服务器的资源使用，本文在 Docker 容器中分别部署了被测试服务器和模拟客户端。在 Docker 工具运行期间，其后台守护进程会在宿主机上创建一个虚拟网桥，用于连接宿主机与各个 Docker 容器的网络。由图可见，被测试服务器与模拟客户端均通过 IP 地址为 172.17.0.1 的虚拟网桥进行通信。

在性能测试过程中，为更真实地模拟受限资源场景，限制被测试服务器最多使用宿主机的 2 个 CPU 核心和 4 GB 内存，以便评估其在有限资源条件下的性能表现和稳定性。

4.2 系统性能测试与功能验证

4.2.1 整体性能测试

性能测试测试环境网络拓扑图如下所示：

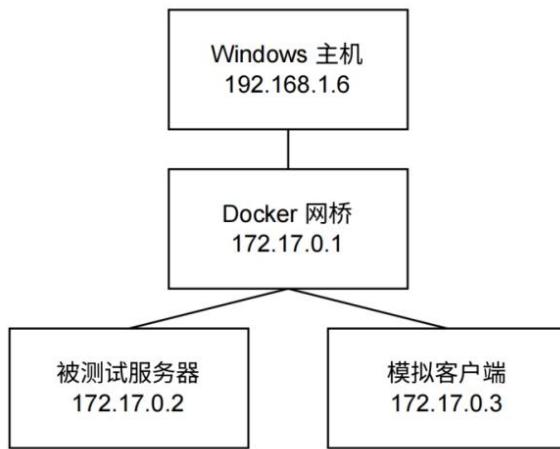


图 4-1 性能测试网络拓扑图

为评估 Azugate 网关系统的性能，选取某登录界面 login.html 作为测试目标。该页面由多个静态资源（如 CSS、JavaScript 等）组成。

该网页的实际界面如下图所示：

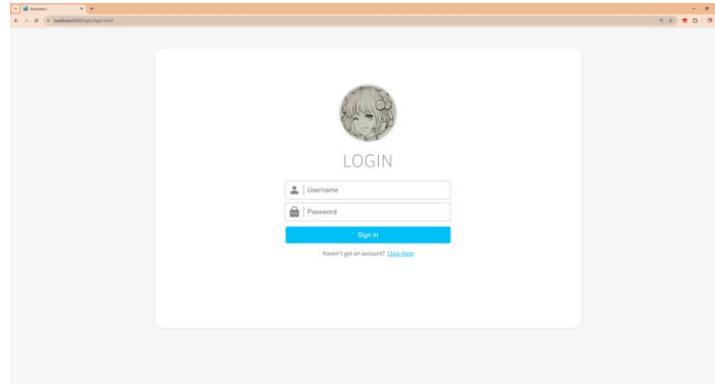


图 4-2 测试网页 login.html

下表给出了该页面所涉及的资源及其大小：

表 4-2 login.html 相关资源与大小

文件路径	文件大小(字节)
login.html	1,122
login.css	1,925
login.js	303
img/account.svg	345
img/passwd.svg	280
Img/avatar.png	856,110

使用 4 线程模式的 wrk 压力测试工具对 Azugate、Nginx、Golang net/http、Python SimpleHTTPServer 和 等常见服务器或框架进行对比测试。由于 PythonSimpleHTTPServer 的实现只支持单线程运行，本节实验将分为单线程与多线程测试。

(1) 在多线程性能测试中，测试对象为 Azugate、Nginx 与 Golang 标准库 net/http。限制程序只能使用两个 CPU 核心，测试得到的结果如图 4-3、图 4-4 与图 4-5 所示：

```
Running 5s test @ http://172.17.0.2:8080/login/login.html
 4 threads and 400 connections
 Thread Stats      Avg      Stdev      Max  +/- Stdev
  Latency    12.24ms   15.97ms  97.26ms   82.26%
  Req/Sec    14.89k     5.52k   33.58k   79.50%
 296142 requests in 5.07s, 340.04MB read
Requests/sec: 58439.94
Transfer/sec: 67.10MB
```

图 4-3 Azugate 压力测试终端输出

```
Running 5s test @ http://172.17.0.2:8080/
 4 threads and 400 connections
 Thread Stats      Avg      Stdev      Max  +/- Stdev
  Latency    13.21ms   16.36ms  65.07ms   81.22%
  Req/Sec    15.48k     4.30k   31.03k   70.50%
 307905 requests in 5.05s, 399.65MB read
Requests/sec: 60924.57
Transfer/sec: 79.08MB
```

图 4-4 Nginx 压力测试终端输出

```
Running 5s test @ http://172.17.0.2:8080/login/login.html
 4 threads and 400 connections
 Thread Stats      Avg      Stdev      Max  +/- Stdev
  Latency    30.26ms   40.34ms 301.62ms   83.16%
  Req/Sec    9.22k     4.18k   26.88k   72.50%
 183590 requests in 5.07s, 229.01MB read
Requests/sec: 36238.87
Transfer/sec: 45.20MB
```

图 4-5 net/http 压力测试终端输出

从图 4-3 与图 4-4 中可得知，Azugate 与 Nginx 性能相近，同时 Azugate 平均延时比 Nginx 少 1 ms。分析原因为 Azugate 使用多线程架构，相较于 Nginx 的多进程架构，线程切换开销更小，有利于提升响应速度和资源利用率。继续分析图 4-5 发现，Golang 标准库 net/http 的性能与平均延时均明显劣于 Azugate 与 Nginx，仅为前者的约一半。其主要原因是 Golang 语言存在 GC（Garbage Collection，垃圾回收）机制^[30]，开发者无法完全掌控内存管理，容易在高并发场景下引发不稳定的性能波动；此外，net/http 的设计相对通用，缺乏对高性能场景的专项优化，例如连接复用策略和事件驱动模型等方面较为简单，难以充分发挥硬件性能，从而导致整体吞吐量和响应延迟不如前两者^[31]。

(2) 在单线程性能测试中，测试对象为 Azugate 与 Python SimpleHTTPServer。限制程序只能使用两个 CPU 核心，测试得到的结果如图 4-6 与图 4-7 所示：

```
Running 1m test @ http://172.17.0.2:8080/login/login.html
 4 threads and 400 connections
 Thread Stats      Avg      Stdev      Max  +/- Stdev
   Latency    12.11ms    8.79ms  100.15ms  70.60%
   Req/Sec     4.07k     1.79k   12.60k  82.57%
 964912 requests in 1.00m, 1.08GB read
  Socket errors: connect 0, read 0, write 0, timeout 1
 Requests/sec: 16057.07
 Transfer/sec: 18.44MB
```

图 4-6 Azugate (单线程) 压力测试终端输出

```
Running 1m test @ http://172.17.0.2:8080/login/login.html
 4 threads and 400 connections
 Thread Stats      Avg      Stdev      Max  +/- Stdev
   Latency    8.19ms    37.32ms   1.76s  99.28%
   Req/Sec   312.99    215.00    1.24k  71.25%
 30136 requests in 1.00m, 37.62MB read
  Socket errors: connect 67, read 0, write 0, timeout 20
 Requests/sec: 501.57
 Transfer/sec: 641.17KB
```

图 4-7 Python SimpleHTTPServer 压力测试终端输出

在单线程模式下，Azugate 的性能仍达到 Python SimpleHTTPServer 的约 30 倍。测试过程中还观察到，Python SimpleHTTPServer 出现了大量连接失败和超时错误。经分析，问题主要源于其采用阻塞式 I/O 架构，无法高效处理并发请求，同时 Python 作为解释型语言，本身在执行效率和系统调用处理上存在瓶颈，难以充分发挥底层硬件性能。

将多线程测试下 Nginx、Azugate、Golang net/http 与单线程测试下 Azugate 与 Python SimpleHTTPServer 的每秒成功处理的请求数 (QPS) 绘制成柱状图，如下所示：

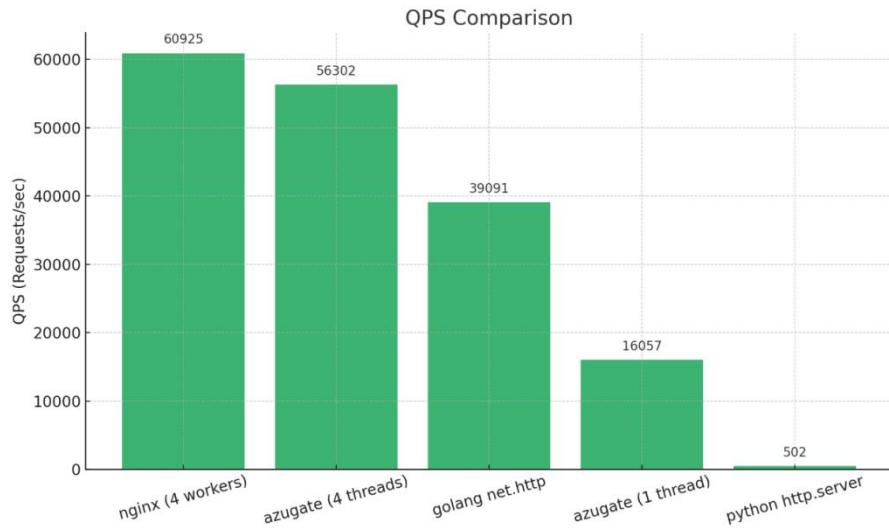


图 4-8 Azugate 与各网络框架的 QPS 对比

在多线程模式下，Azugate 的 QPS 达到工业级网关 Nginx 的约 92%，为 Golang net/http 库的约 144%。在单线程场景中，Azugate 的性能为 Python SimpleHTTPServer 的约 30 倍，展现出显著的性能优势。

4.2.2 核心功能验证

(1) 应用代理示例

本小节旨在测试网关对实际应用的 HTTP 和 WebSocket 协议的代理能力。被测试代理的服务 Bytebase 是一款数据库安全管理平台，其内置的 SQL 编辑器功能通过 WebSocket 实现代码自动补全，而其服务接口基于 HTTP 协议构建。

本测试网络拓扑如下图所示：

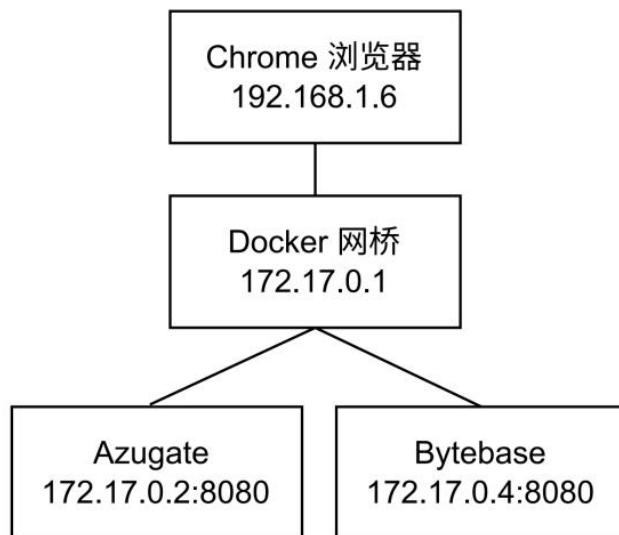


图 4-9 Azugate 实际应用测试网络拓扑图

测试流程与结果如图 4-10、图 4-11 与图 4-12 所示：

```

localhost:8051 | ConfigService / ConfigRouter
Message Authorization Metadata Service definition Scripts Settings

1 "routers": [
2   {
3     "source": "/lsp",
4     "destination": "172.17.0.4:8080/lsp"
5   },
6   {
7     "source": "*",
8     "destination": "172.17.0.4:8080"
9   }
10 ]
11
12
  
```

图 4-10 调用 gRPC 接口设置代理路由

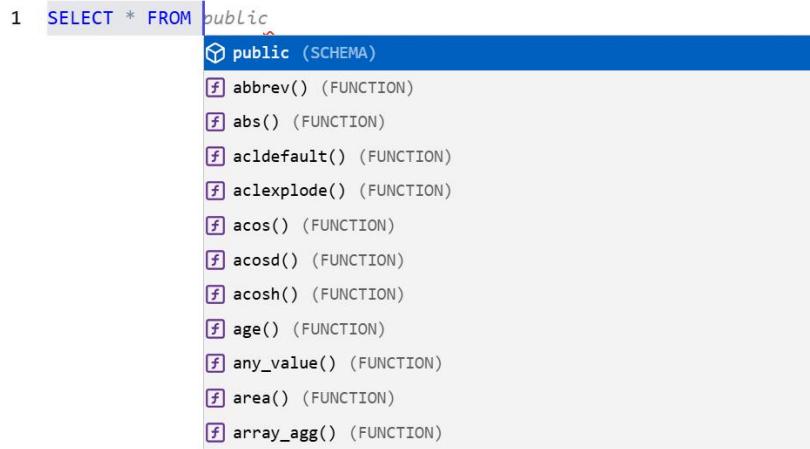


图 4-11 Bytebase SQL 自动补全

```
97:16:50 | azugate.cc:276 | loading config from ../resources/config.yaml
97:16:50 | azugate.cc:277 | Health check will be performed every 3 seconds
97:16:50 | azugate.cc:278 | azugate is listening on port 8081
97:16:50 | azugate.cc:279 | server is running with 1 thread(s)
97:16:50 | azugate.cc:280 | gRPC server is listening on port 50051
97:16:50 | azugate.cc:282 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:16:50 | azugate.cc:283 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.OrgPolicyService/GetPolicy
97:16:50 | azugate.cc:285 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.SQLService/Check
97:16:50 | azugate.cc:287 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.SQLService/SearchQueryHistories
97:16:50 | azugate.cc:289 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.SQLService/Query
97:16:50 | azugate.cc:291 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.DatabaseService/ListDatabases
97:16:50 | azugate.cc:293 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.ProjectService/BatchGetIamPolicy
97:16:50 | azugate.cc:295 | http proxy 172.17.0.1 -> 172.17.0.4:8080/bytebase.v1.WorksheetService/SearchWorksheets
97:16:50 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:16:53 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:16:56 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:16:59 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:17:02 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
97:17:05 | azugate.cc:299 | ws proxy 172.17.0.1 -> 172.17.0.4:8080/lsp
```

图 4-12 Azugate 代理应用终端输出

可以看到，Azugate 根据 gRPC 配置正确完成了路由设置。同时，从图 4-12 中可以看出，网关成功代理了应用程序的 HTTP 与 WebSocket 请求。其中，Bytebase 的 SQL 自动补全功能通过 WebSocket 进行心跳检测，因此在图像底部可以看到 Azugate 持续代理 WebSocket 数据的记录。这些重复出现的代理请求，即为 Bytebase 以每 3 秒一次的固定间隔发送的心跳数据。

(2) HTTP 内容压缩与 TLS 连接测试

本实验通过 curl 工具模拟客户端访问场景，借助 Azugate 的 gRPC 管理接口动态启用或关闭 HTTP 内容压缩与 TLS 支持，以评估网关在安全性与传输效率方面的灵活配置能力。测试结果如下图 4-13、图 4-14 与图 4-15 所示：

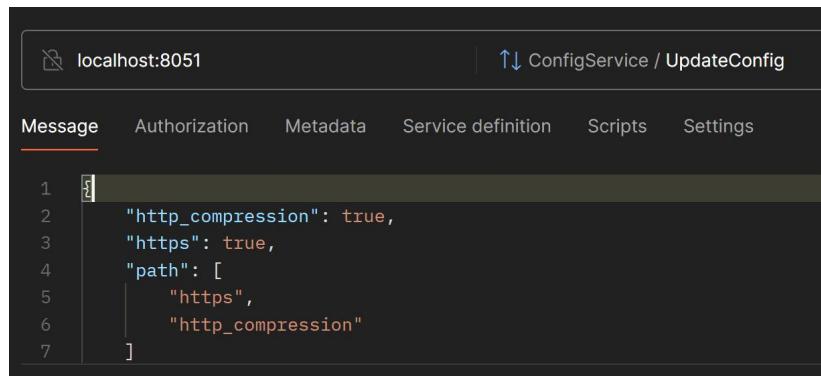


图 4-13 使用 gRPC 管理接口开启 TLS 与内容压缩

```

root@84ffff98961cc:~/Codes/azugate/build# curl -v http://localhost:8080/welcome.html
* Host localhost:8080 was resolved.
* IPv6: ::1
* IPv4: 127.0.0.1
* Trying [:1]:8080...
* connect to ::1 port 8080 from ::1 port 59724 failed: Connection refused
* Trying 127.0.0.1:8080...
* Connected to localhost (127.0.0.1) port 8080
* using HTTP/1.x
> GET /welcome.html HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.13.0
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Content-Type:text/html
< Connection:Close
< Content-Length:230
<
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>azugate</title>
</head>

<body>
    <h1>Welcome to Azugate...</h1>
</body>

```

图 4-14 未开启 TLS 与内容压缩时客户端得到的响应

```

* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384 / x25519 / RSASSA-PSS
* ALPN: server did not agree on a protocol. Uses default.
* Server certificate:
*   subject: C=AU; ST=Some-State; O=Internet Widgits Pty Ltd
*   start date: Dec 10 08:57:50 2024 GMT
*   expire date: Mar 15 08:57:50 2027 GMT
*   subjectAltName: host "localhost" matched cert's "localhost"
*   issuer: C=AU; ST=Some-State; O=Internet Widgits Pty Ltd
*   SSL certificate verify ok.
*   Certificate level 0: Public key type RSA (2048/112 Bits/secBits), signed us
*   Certificate level 1: Public key type RSA (2048/112 Bits/secBits), signed us
* Connected to localhost (127.0.0.1) port 8080
* using HTTP/1.x
> GET /welcome.html HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.13.0
> Accept: */*
> Accept-Encoding: deflate, gzip, br, zstd
>
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* Request completely sent off
< HTTP/1.1 200 OK
< Content-Type:text/html
< Connection:Close
< Content-Encoding: gzip
< Transfer-Encoding: chunked

```

图 4-15 未开启 TLS 与内容压缩时客户端得到的响应

在 HTTP 内容压缩以及 TLS 未开启之前，图 4-14 显示 curl 访问网关没有建立 TLS 的过程，同时返回的 HTTP 首部并不包含“Content-Encoding: gzip”等字样。而在图 4-15 中详细的包含了 TLS 连接建立过程，其中“Content-Encoding: gzip”与“Transfer-Encoding: chunked”信息说明了网关当前正在启用了 HTTP 内用压缩以及分块传输优化。

(3) 速率控制

为验证 Azugate 的速率控制能力，本实验使用 wrk 工具对“Welcome to Azugate”页面发起高并发请求。通过调用 Azugate 提供的 gRPC 管理接口，对比启用前后的请求处理速率，测试结果如下图 4-16 至图 4-20 所示：

```
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
[info] 41390 | 04/13/25 18:08:09 | azugate.cc:91 | connection from 127.0.0.1
```

图 4-16 未开启速率控制时 Azugate 的终端输出

```
Running 5s test @ http://localhost:8080/welcome.html
 1 threads and 1 connections
 Thread Stats      Avg      Stdev      Max    +/- Stdev
   Latency        2.67ms    10.22ms   74.87ms   93.50%
   Req/Sec       9.10k     2.29k    13.60k    76.47%
 46153 requests in 5.10s, 13.69MB read
 Requests/sec: 9049.49
 Transfer/sec: 2.68MB
```

图 4-17 未开启速率控制时 Azugate 的压力测试结果

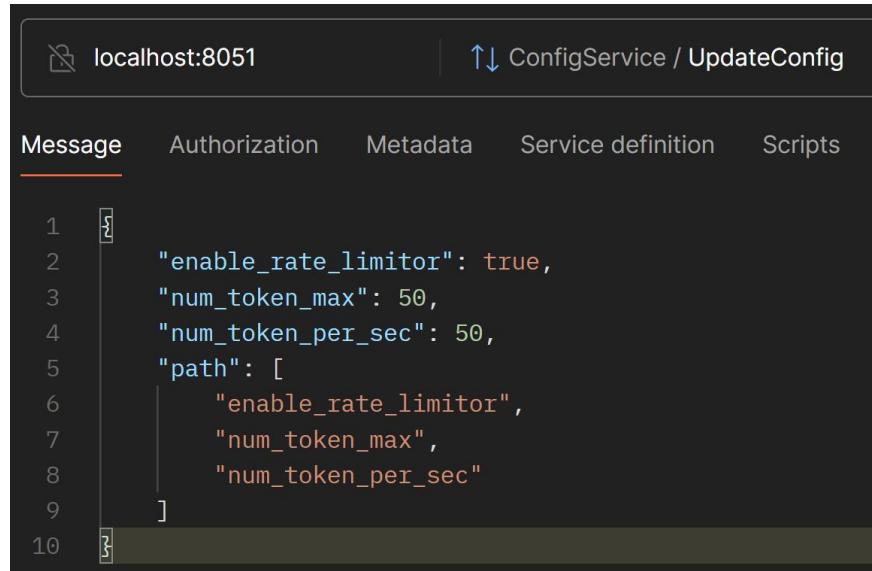


图 4-18 调用 gRPC 接口开启速率控制

```
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
[info] 41390 | 04/13/25 18:05:26 | azugate.cc:91 | connection from 127.0.0.1
[warning] 41390 | 04/13/25 18:05:26 | dispatcher.cc:61 | request rejected by rate limiter
```

图 4-19 开启速率控制时 Azugate 的终端输出

```

Running 5s test @ http://localhost:8080/welcome.html
 1 threads and 1 connections
 Thread Stats      Avg      Stdev     Max   +/- Stdev
  Latency    72.11us  32.55us 252.00us  87.63%
  Req/Sec   169.57    197.37 494.00    71.43%
 299 requests in 5.00s, 90.81KB read
  Socket errors: connect 0, read 44491, write 0, timeout 0
Requests/sec:      59.76
Transfer/sec:    18.15KB

```

图 4-20 开启速率控制时 Azugate 的压力测试结果

实验过程中，Azugate 通过 gRPC 管理接口对目标路径动态配置了速率限制。在未启用限速功能时，从图 4-16 中 Azugate 的终端输出可以观察到它正常地接受了来自本地的 TCP 连接请求，同观察图 4-17，Azugate 的平均每秒处理请求数达到了 9049.49 个。

随后，通过 gRPC 接口配置速率限制策略：每秒生成 50 个令牌，最大可堆积令牌数为 50。从图 4-20 可见，使用 wrk 工具进行为期 5 秒的压测，Azugate 共处理了 299 个请求。同时从图 4-19 网关终端输出信息可以观察到，Azugate 对于超过令牌桶容量限制的请求进行了丢弃处理，并输出了“request rejected by rate limiter”的警告信息。

根据配置，Azugate 在 5 秒内可生成 250 个令牌（ 5×50 ），加上最大堆积的 50 个令牌，理论上每秒最多可处理 300 个请求。实际结果与理论值高度一致，验证了 Azugate 的限速机制正常工作。

(4) 负载均衡

本小节通过 Azugate 的 gRPC 管理接口，动态添加 HTTP 路由配置，实现将对路径“/”的访问请求，平均分发至本地不同端口上的三个 Python 网页代理服务器。负载均衡实验网络环境如下图所示：

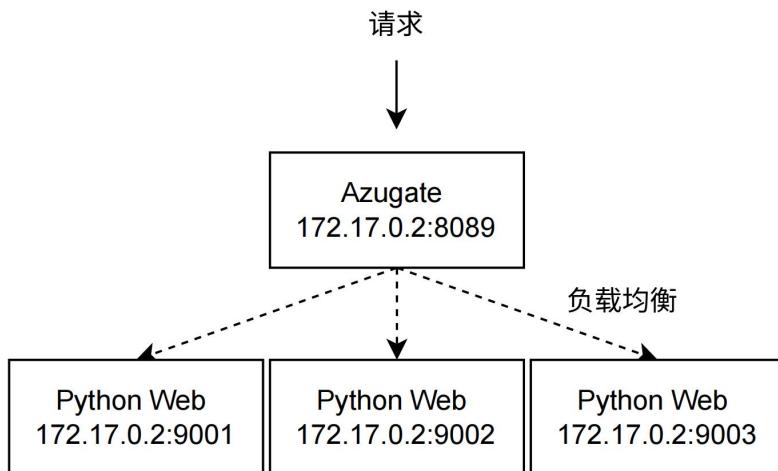


图 4-21 负载均衡实验网络环境

使用 wrk 测试工具对网关 8089 端口发起压力测试，测试结果如下图所示：

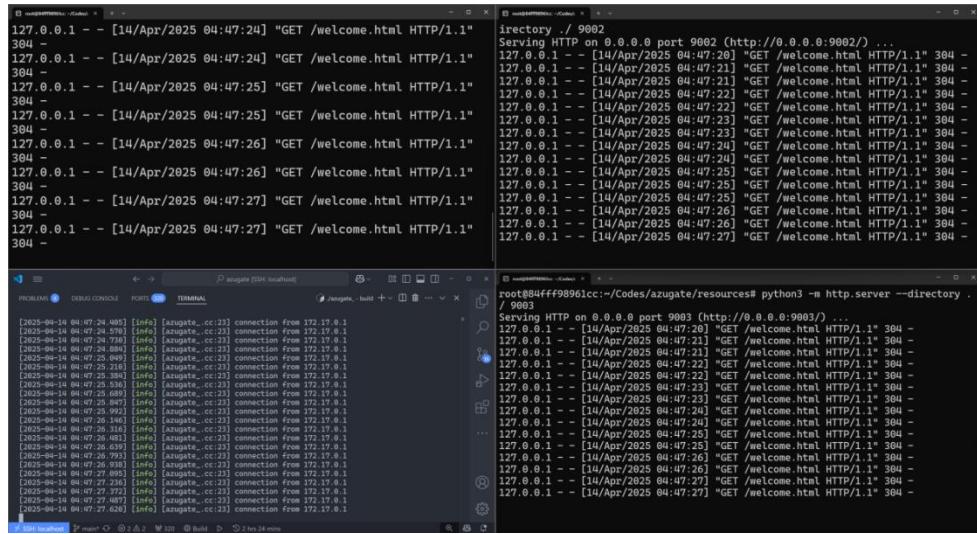


图 4-22 负载均衡实验结果

图中左下角终端为网关运行的输出，其他三个终端分别代表运行在端口 9001（左上）、9002（右上）、9003（右下）的 Python 网页服务器。由图可以看输出实验开始后网关收到的 HTTP 请求被均匀的分配到了上游的网页服务器中。

(5) IP 防火墙

为验证 Azugate 的 IP 防火墙功能，本实验使用其 gRPC 管理接口动态添加指定黑名单 IP，并观察被封禁用户访问资源时的响应情况。通过对比设置前后的请求结果，评估访问控制机制的有效性，实验过程如下图 4-23 至图 4-26 所示：

```
[INFO] 32273 | 04/13/25 17:04:11 | azugate.cc:144 | loading config from ../resources/config.yaml
[INFO] 32273 | 04/13/25 17:04:11 | azugate.cc:178 | azugate is listening on port 8080
[INFO] 32273 | 04/13/25 17:04:11 | azugate.cc:191 | server is running with 1 thread(s)
[INFO] 32274 | 04/13/25 17:04:11 | azugate.cc:171 | gRPC server is listening on port 50051
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:23 | azugate.cc:91 | connection from 172.17.0.1
[INFO] 32275 | 04/13/25 17:04:47 | azugate.cc:91 | connection from 172.17.0.1
```

图 4-23 未限制 IP 访问时终端输出



图 4-24 通过 gRPC 管理接口限制 IP 访问

```
[warning] 32275 | 04/13/25 17:04:54 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:05:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:05:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:05:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:05:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:05:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:05:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:06:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:06:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:06:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:06:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:06:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:06:24 | filter.cc:14 | reject connection from 172.17.0.1
[info] 32275 | 04/13/25 17:06:24 | azugate.cc:91 | connection from 172.17.0.1
[warning] 32275 | 04/13/25 17:06:24 | filter.cc:14 | reject connection from 172.17.0.1
```

图 4-25 限制 IP 访问时终端输出



该网页无法正常运作

localhost 未发送任何数据。

ERR_EMPTY_RESPONSE

重新加载

图 4-26 限制 IP 访问时的浏览器界面

在 IP 黑名单设置之前，172.17.0.1 地址的请求可以正常访问网关资源。而在图 4-24 的配置后，网关拦截了来自 172.17.0.1 地址的请求，并在终端输出了相应的警告信息。用户端使用 Chrome 浏览器会查看到连接被终止的提示。

(6) OAuth 外部认证

为验证 Azugate 的外部认证功能，实验构建了一个以 Auth0 为身份验证服务的环境。用户首次访问受保护资源时将被重定向至认证界面，认证通过后返回所请求资源页面“Welcome to Azugate”。本实验的网络拓扑如 3.3.3 小节中图 3-14 所示。外部认证测试过程如图 4-27 至图 4-30 所示：

```
localhost:8051
↑↓ ConfigService / UpdateConfig

Message Authorization Metadata Service definition Scripts Settings

1 "enable_external_auth": true,
2 "client_secret": "9ZpguaFYAlJtjuJmC6qmriY9IiUzPZm9IKgPhWL8pL78aMLL_zaaGsIPansIoIu",
3 "client_id": "wKWP9oGUySyMYDuISRTZ5EeGeB8S0l6F",
4 "auth_domain": "dev-uep626a5ecgmzopx.us.auth0.com",
```

图 4-27 使用 gRPC 接口配置 OAuth 外部认证

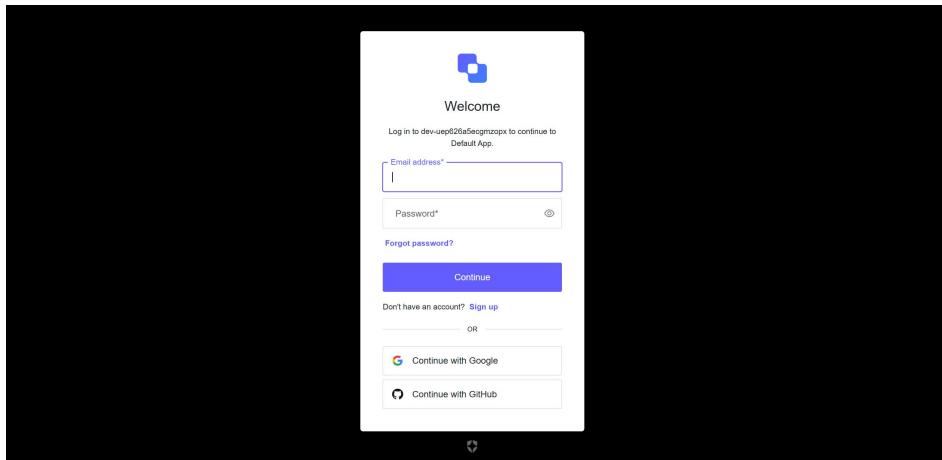


图 4-28 Auth0 外部认证界面

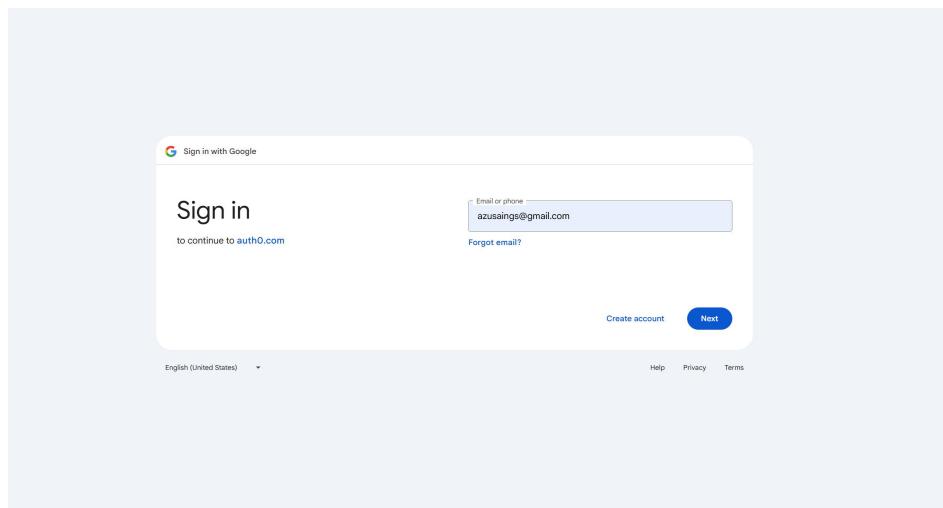


图 4-29 使用 Auth0 支持的登陆方式进行登录

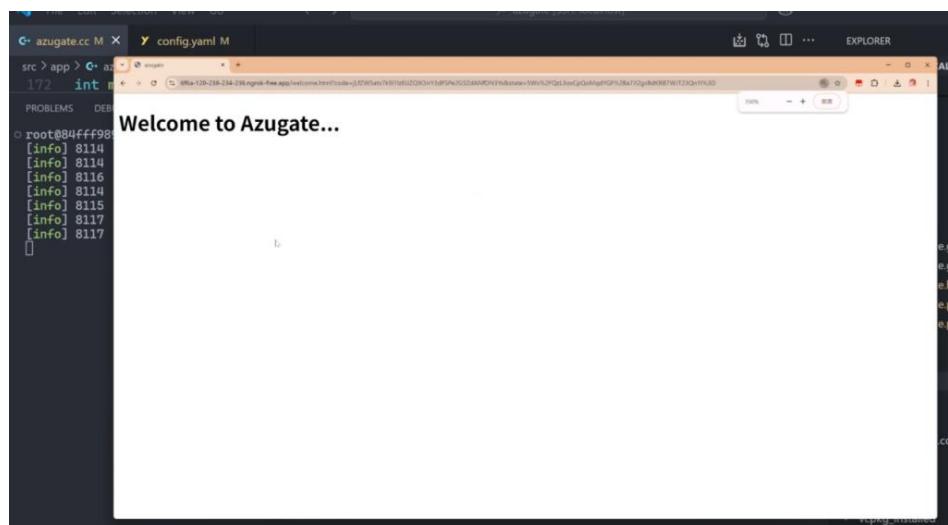


图 4-30 成功登陆后网管返回静态资源

使用 gRPC 管理接口将 OAuth 提供者（OAuth Provider）的客户端标识符、客户端密钥与回调地址设置好后，访问网关代理的资源将先被网关重定向到认

证页面。本实验使用的 OAuth 提供商 Auth0 支持 Google、Github 以及 Microsoft 等多种登陆方式。所以测试过程中可以使用 Google 账户进行登录。成功登录后用户能正常访问所需资源。在本实验中模拟用户正常登录后将会看到 Azugate 默认欢迎界面“Welcome to Azugate”。

(7) 心跳检测

为验证 Azugate 的心跳检测功能，本小节构建了包含多个后端服务节点的代理集群，使用 Postman 工具模拟应用向 Azugate 的 gRPC 管理接口发送注册心跳检测，接着通过关闭其中一个节点的方式模拟服务故障。Azugate 能够在检测到节点不可达后，及时进行预警，从而保障系统的可用性与稳定性。

使用与图 4-21 中类似的测试环境，实验开始前先启动网关以及三个待进行心跳检测的应用服务器，并通过调用 gRPC 管理接口用三个待进行心跳检测服务的 IP 地址热更新网关内部配置。

启动网关以及应用服务器与进行 gRPC 管理接口设置流程如下图 4-31 与图 4-32 所示：

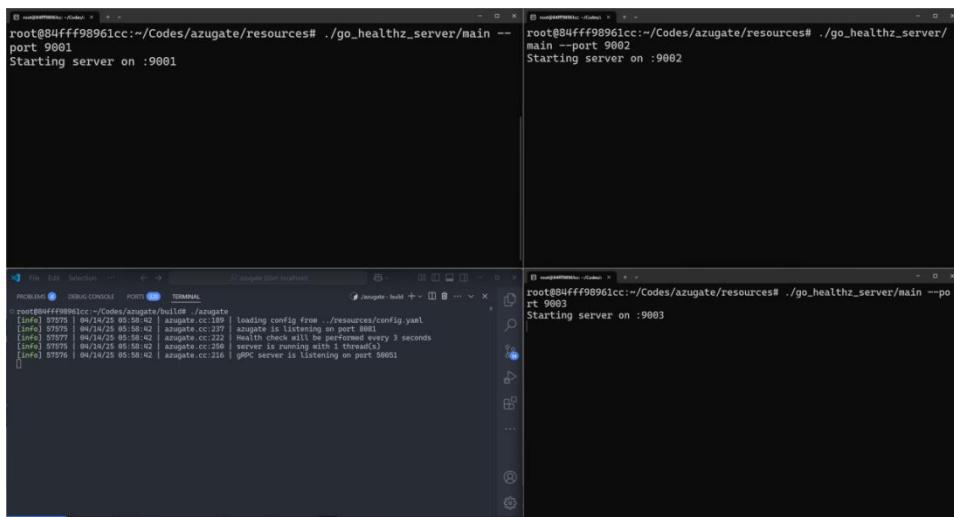


图 4-31 启动网关以及三个被心跳检测的服务器

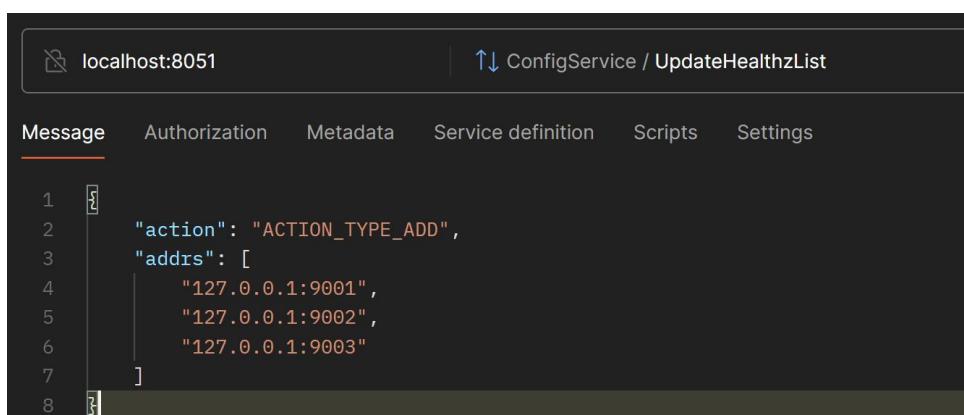


图 4-32 使用 gRPC 管理接口添加心跳检测地址列表

实验开始后网关每 3 秒会对地址列表中的服务进行健康检查，实验过程中各终端输出如下图 4-33 与图 4-34 所示：

```
root@84fff98961cc:~/Codes/azugate/resources# ./go_healthz_server/main --port 9001
Starting server on :9001
new connection
new connection
new connection
new connection
new connection

root@84fff98961cc:~/Codes/azugate/resources# ./go_healthz_server/main --port 9002
Starting server on :9002
new connection
new connection
new connection
new connection
new connection

root@84fff98961cc:~/Codes/azugate/build$ ./azugate
[Info] 59795 | [2023/04/14 05:58:42] azugate.cc:221 loading config from: .../resources/config.yaml
[Info] 59795 | [2023/04/14 05:58:42] azugate.cc:227 azugate is listening on port 8081
[Info] 59797 | [2023/04/14 05:58:42] azugate.cc:227 Health check will be performed every 3 seconds
[Info] 59798 | [2023/04/14 05:58:42] azugate.cc:228 server is running with 1 thread(s)
[Info] 59798 | [2023/04/14 05:58:42] azugate.cc:298 gRPC server is listening on port: 9091

root@84fff98961cc:~/Codes/azugate/resources# ./go_healthz_server/main --port 9003
Starting server on :9003
new connection
new connection
new connection
new connection
```

图 4-33 网关每 3 秒向服务发送心跳检测请求

图 4-34 服务终止后网关发出预警

由图 4-33 观察可知注册心跳检测服务之后，各个网页服务器定时接收到了来自网关的检查请求。由图 4-34 观察可发现在人为终止运行在端口 9002（右上）与端口 9003（右下）上的网页服务器后，网关（左下）终端输出地址为“127.0.0.1:9002”与“127.0.0.1:9002”的服务不可用的警告。

第 5 章 总结与展望

本论文围绕 Azugate 网关系统的设计与实现展开，重点探讨其在基础代理功能与云原生架构适配方面的关键技术路径与性能优化策略。

在基础代理功能方面，Azugate 实现了对 TCP、HTTP 及 WebSocket 协议的统一代理支持，并具备 TLS 连接的协商与建立能力。系统自研的高性能协议解析器，统一处理 HTTP 与 WebSocket 协议，有效提升了协议栈处理的一致性与效率。

为进一步优化转发性能与资源利用率，Azugate 引入了包括零拷贝传输机制、基于 Gzip 的 HTTP 内容压缩策略，以及基于前缀树结构的高效路由匹配算法等多项关键技术手段。此外，系统还集成了速率控制、负载均衡与 IP 层防火墙等核心能力，显著增强了其在复杂网络环境中的适应性与扩展性。

在云原生环境适配方面，Azugate 提供基于 gRPC 的动态管理接口，支持配置热更新；兼容 OAuth 协议以对接外部认证服务器，实现统一访问控制；通过内置心跳机制提升故障感知与服务可用性；同时具备良好的容器化部署能力，便于与微服务体系及服务网格架构深度集成。

性能评估结果表明，Azugate 在多线程环境下的请求处理能力达到 Go net/http 的 144%、Nginx 的 92%；在单线程场景下，其性能为 Python HTTPServer 的 30 倍，充分验证了其在高并发负载下的工程实用性和性能潜力。

尽管 Azugate 在性能和功能方面展现出良好表现，但与成熟网关相比仍存在以下较明显的不足：

(1) 缺乏缓存机制：系统尚未设计静态资源缓存或反向代理缓存层，无法在高频访问场景中减轻后端压力，影响整体响应效率。这也是 Azugate 与 Nginx 性能差距的一大重要原因。

(2) 不支持 HTTP 多路复用与长连接：当前仅实现 HTTP/1.1 标准，缺乏对 HTTP/2 等现代协议的支持，也未启用 Keep-Alive 等连接复用机制，不利于高并发场景下的资源利用率。

(3) 跨平台优化支持不足：系统在性能优化中使用了如 sendfile() 这样的 Linux 特定系统调用，在提升传输效率的同时，也限制了系统在其他平台上的可移植性。目前尚未针对不同操作系统提供替代的优化方案，例如在 Windows 中可使用“TransmitFile”，在 macOS 中可使用“sendfile”的 BSD 实现等。

综上，Azugate 虽在基本功能和性能表现上已具备一定竞争力，但若要发展为可应用于实际生产环境的高性能网关系统，仍需在跨平台性、协议支持、性能优化等多个方面进行系统性提升。

参考文献

- [1] 丁玉婷. 面向微服务的 API 网关流量控制与调度系统设计与实现 [D]. 华中科技大学, 2024.
- [2] 架构驿站 Luga Lee. 一文读懂云原生网关演进史 [J]. 大数据时代, 2024, (11):11-15.
- [3] 刘玉婷, 吴彤, 任祥辉. 基于 Nginx 负载均衡的大规模脑电信号处理集群架构设计与实现 [J]. 信息通信技术与政策, 2025, 51(03):16-24.
- [4] 马飞, 杜岚, 仇保琪. 云原生时代身份管理方法研究 [J]. 保密科学技术, 2024, (10):34-41
- [5] Wang X , Zhao H , Zhu J . GRPC: A Communication Cooperation Mechanism in Distributed Systems [J]. ACM SIGOPS Operating Systems Review, 1993, 27(3):75-86. DOI: 10.1145/155870.155881:75-76.
- [6] Stevens W. R. , Fenner B. , Rudoff A. M. UNIX Network Programming, Volume 1: The Sockets Networking API (3rd Edition) [M]. Addison-Wesley, 2004.
- [7] 游双. Linux 高性能服务器编程 [M]. 机械工业出版社, 2013.
- [8] AndrewS. Tanenbaum. Modern operating systems:2nd edition-英文版 [M]. 机械工业出版社, 2002.
- [9] Stevens W R . Advanced Programming in the UNIX Environment [M]. Posts & Telecom Press, 1992.
- [10] Kurose J F , Ross K W . Computer Networking: A Top-Down Approach Featuring the Internet [M]. Addison-Wesley, 2002.
- [11] Knuth D E . Preface to The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd Edition [M]. [2025-04-15].
- [12] 谢希仁. 计算机网络 [M]. 电子工业出版社, 2008.
- [13] Frier A . The SSL 3.0 Protocol [J]. Netscape Communication Corp, 1996:41-42.
- [14] R. Fielding, J. Gett S, J. Mogul. RFC2616 – Hypertext transfer protocol - HTTP/1.1 [J]. Computer Science & Communications Dictionary, 1999. DOI:10.1109/GLOCOM.2003.1258974.
- [15] Belshe M , Peon R , Thomson M . Hypertext Transfer Protocol Version 2 (HTTP/2) [R]. Politics & Gender, 2015, 1(12):189-192. DOI:10.1017/S1743923X05232014.
- [16] Chatzoglou E , Kouliaridis V , Kambourakis G , et al. A hands-on gaze on HTTP/3 security through the lens of HTTP/2 and a public dataset [J]. Comput. Secur. 2022, 125:103051. DOI:10.1016/j.cose.2022.103051:5-6.
- [17] 美 布赖恩特 R. E, 美 Bryant Randal E, 美 O'Hallaron David. 深入理解计算机系统 [M]. 中国电力出版社, 2004.
- [18] DavidGourley. HTTP 权威指南 [M]. 人民邮电出版社, 2012.
- [19] Fette I , Melnikov A . RFC 6455 – The WebSocket Protocol [R]. 2011.
- [20] Josuttis N M . The C++ standard library: a tutorial and reference [M]. Addison-Wesley Longman Publishing Co. Inc. 1999.
- [21] Sakimura N , Bradley J . The authorization request in RFC6749 utilizes query

- parameter serialization. This specification defines the authorization request using JWT serialization. The request is sent by value through[R]. 2015.
- [22]Tsai W T , Sun X , Balasooriya J . Service-Oriented Cloud Computing Architecture[C]//Seventh International Conference on Information Technology: New Generations. IEEE Computer Society, 2010. DOI:10.1109/ITNG.2010.214.
- [23]Nichols K . Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers[R]. RFC2474, 1998.
- [24]Beyer B , Jones C , Petoff J , et al. Site Reliability Engineering: How Google Runs Production Systems[M]. O'Reilly Media, Inc. 2016.
- [25]Zheng T , Tang R , Chen X , et al. KubeFuzzer: Automating RESTful API Vulnerability Detection in Kubernetes[J]. Computers, Materials & Continua, 2024, 81(1). DOI:10.32604/cmc.2024.055180:1596:1597.
- [26]George Coulouri, Jean Dollimore, Tim Kindberg, 等. 分布式系统概念与设计[M]. 计算机教育, 2004(10).
- [27]Brendan Gregg. 性能之巅: 洞悉系统、企业与云计算[M]. 电子工业出版社, 2015.
- [28]Bernstein, David. Containers and Cloud: From LXC to Docker to Kubernetes[J]. IEEE cloud computing, 2014:82–83.
- [29]于烨;李斌;刘思尧;. Docker 技术的移植性分析研究[J]. 软件, 2015(07):58–60.
- [30]Dijkstra E W , Lamport L , Martin A J , et al. On-the-fly Garbage Collection: An Exercise in Cooperation[J]. Communications of the ACM, 1975, 21(11):966–975. DOI:10.1145/359642.359655.
- [31]Donovan A A A, Kernighan B W. The Go Programming Language[M]. Boston: Addison-Wesley, 2015.

致谢

随着毕业设计的顺利完成，本科阶段的学习也接近尾声。在这段充实的学习旅程中，我收获了许多宝贵的经验，深刻体会到理论与实践相结合的重要性。感谢所有在这过程中给予我帮助的人。

首先，我要衷心感谢我的指导老师——任子良老师。在整个毕业设计的过程中，任老师的耐心指导和悉心帮助使我受益匪浅。无论是选题、设计，还是调试过程中遇到的难题，任老师都给了我宝贵的意见和建议，帮助我不断完善和提升项目。没有任老师的 support，我无法顺利完成这篇论文。此外，我要感谢我的同学们，在这段时间里，我们互相讨论、分享经验，他们的鼓励和帮助让我更加坚定了完成项目的信心。感谢他们在我遇到瓶颈时给予的支持。

在项目开发过程中，C++ 语言的复杂性和高性能内存管理带来了很多挑战，尤其是在异步编程模型中的对象生命周期管理、调试，以及在实现过程中需要逐字节处理数据的艰苦过程，此外，网络协议的实现也增加了不少难度。为了解决这些问题，我查阅了大量技术文档和书籍，学习了相关的调试技巧，并深入研究了具体的实现方法。经过不懈努力，我逐步克服了这些技术难题，确保了项目的顺利进行。

未来，我将继续深入学习网络编程和底层技术，争取在技术领域取得更大进步。