QUIC 技术白皮书

Copyright © 2026 新华三技术有限公司 版权所有，保留一切权利。
非经本公司书面许可，任何单位和个人不得擅自摘抄、复制本文档内容的部分或全部，并不得以任何形式传播。
除新华三技术有限公司的商标外，本手册中出现的其它公司的商标、产品标识及商品名称，由各自权利人拥有。
本文中的内容为通用性技术信息，某些信息可能不适用于您所购买的产品。

目 录

1 QUIC 协议概述 ···· 1
2 QUIC 协议产生背景 ···· 1
2.1 HTTP 协议的发展 ···· 1
2.1.1 HTTP 协议的演进过程 ···· 1
2.1.2 HTTP 协议的关键突破 ···· 2
2.2 TCP 协议的性能瓶颈 ···· 3
2.2.1 队头阻塞 ···· 4
2.2.2 高握手延迟 ···· 6
2.2.3 连接迁移的缺失 ···· 8
2.2.4 加密与安全的局限性 ···· 8
2.2.5 拥塞控制的保守性 ···· 9
3 QUIC 协议技术实现 ···· 9
3.1 QUIC 协议架构 ···· 9
3.1.1 QUIC 协议栈 ···· 9
3.1.2 QUIC 协议报文格式 ···· 10
3.2 QUIC 协议的性能优化 ···· 12
3.2.1 显著降低握手延迟 ···· 12
3.2.2 增强型可靠传输 ···· 14
3.2.3 高效的传输层多路复用 ···· 15
3.2.4 灵活的流量控制 ···· 15
3.2.5 智能化的拥塞控制 ···· 17
3.2.6 支持无缝连接迁移 ···· 18
4 TCP 与 QUIC 实现机制对比总结 ···· 18
5 典型应用 ···· 19
5.1 QUIC 在 SDWAN 网络中的应用 ···· 19
5.2 QUIC 在 VDI 网关上的应用 ···· 20
6 参考文献 ···· 21

1 QUIC 协议概述

QUIC（Quick UDP Internet Connections）是由 Google 主导设计、后由 IETF 标准化的新一代传输层协议，旨在通过用户数据报协议（UDP）实现高效、安全的互联网传输。作为 TCP+TLS+HTTP/2 组合的替代方案，QUIC 深度融合了连接管理、加密传输、多路复用等核心特性，显著降低了连接延迟，提升了网络拥塞控制能力，并为移动互联网与高丢包环境提供了优化支持。

2 QUIC 协议产生背景

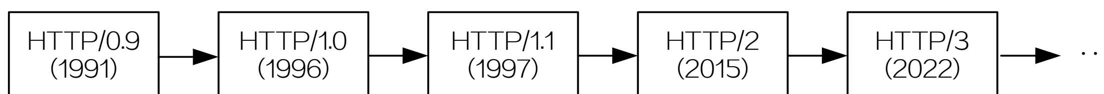
QUIC 的诞生源于传统互联网协议（如 TCP 和 HTTP）在移动互联网、高延迟网络及复杂传输场景中的局限性。本章将从 HTTP 协议的演进和 TCP 的固有缺陷两方面展开，分析 QUIC 解决这些问题的必要性。

2.1 HTTP协议的发展

2.1.1 HTTP 协议的演进过程

HTTP（HyperText Transfer Protocol，超文本传输协议）是互联网上应用最为广泛的协议之一，它定义了客户端与服务器之间的通信规则。

图1 HTTP 协议演进过程



HTTP 协议从 0.9 到 3.0 的演进反映了互联网对更高效率、更低延迟和更强可靠性的持续追求：

(2) 最初的 HTTP/0.9 极其简单，仅支持 GET 方法获取纯文本 HTML，没有头部、状态码或会话管理，功能极为受限。

(3) HTTP/1.0 引入了请求/响应头部，支持 Content-Type 等元数据，新增 POST、HEAD 等方法，并采用状态码规范响应处理，可传输文本、图片等多种数据类型。但它的主要缺陷是每次请求都需新建 TCP 连接，导致高延迟和服务器负担。

(4) HTTP/1.1 优化了连接效率，默认采用持久连接（复用 TCP 连接）和管道化（允许并行发送请求），并引入分块传输编码和精细缓存控制（如 Cache-Control）。然而，队头阻塞（前序请求延迟影响后续请求）和头部冗余（未压缩的重复头部）仍限制性能。

(5) HTTP/2 彻底革新了通信模式，采用二进制分帧替代文本传输，支持多路复用（真正并行请求）、头部压缩（HPACK）和服务器推送，大幅提升传输效率。但由于 HTTP/2 仍依赖 TCP，其 TCP 队头阻塞问题在弱网环境下依然存在。

(6) HTTP/3 进一步优化，基于 QUIC 协议（UDP 实现），彻底摆脱 TCP 限制，减少握手延迟，并增强多路复用和连接迁移能力，使 HTTP 协议在移动网络和高延迟环境下表现更优。

2.1.2 HTTP 协议的关键突破

从 HTTP/0.9 到 HTTP/3 的演进过程中，核心优化方向始终聚焦于连接效率、并行传输和头部压缩三大领域。HTTP/2 通过二进制分帧、多路复用和头部压缩实现了重大突破，但由于其基于 TCP 的设计无法与 QUIC 协议兼容，IETF 最终推出了 HTTP/3 标准以充分发挥 QUIC 的先进特性。

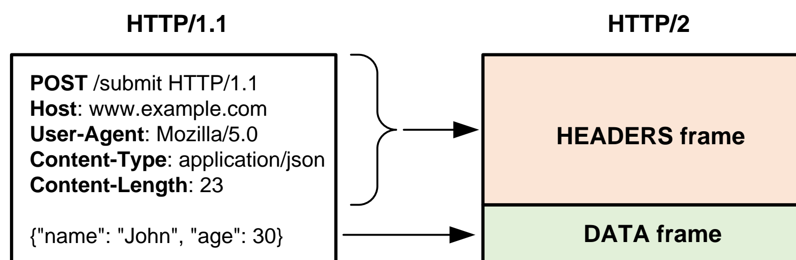
HTTP/3 在应用层完全继承了 HTTP/2 的核心优势，包括：

- 相同的二进制分帧机制
- 基于流的多路复用体系
- 高效的头部压缩方案

1. 二进制分帧

HTTP/1.1 采用纯文本格式传输请求和响应，缺乏长度标记和唯一标识，必须严格按顺序处理，容易导致严重的队头阻塞（HOL Blocking）问题。HTTP/2 引入的革命性二进制分帧机制将消息拆分为独立的帧（如 HEADERS 帧、DATA 帧），每个帧包含精确的长度字段和唯一的流 ID，支持乱序传输和按流重组，在应用层彻底解决了队头阻塞问题。

图2 HTTP 消息



2. 多路复用

HTTP/1.1 虽然支持在单个 TCP 连接上发送多个请求（长连接），但顺序处理机制仍存在队头阻塞。浏览器被迫采用两种补救方案：

- 域名分片：将资源分散到多个子域名建立独立 TCP 连接。
- 多 TCP 连接：对同一域名建立多个并行连接（通常 6~8 个）。

这些方案虽提升并发能力，却显著增加了连接管理开销。而 HTTP/2 通过创新的多路复用机制完美解决了这个问题，其核心设计基于三个关键概念。

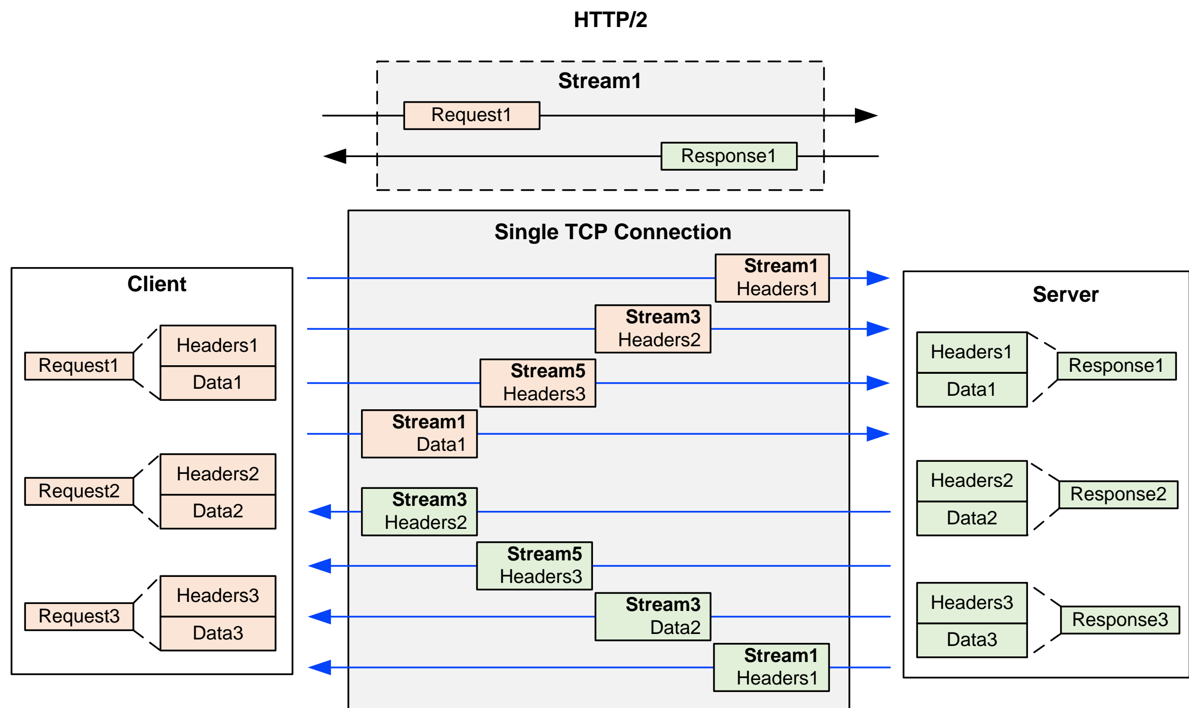
表1 HTTP/2 多路复用架构要素

概念	定义	特点
连接（Connection）	客户端和服务器之间的单个TCP连接	承载多个双向流；支持多路复用
流（Stream）	连接中的逻辑通道，对应一个完整的请求和响应周期	双向通信；支持优先级控制；客户端发起流使用奇数ID，服务器发起流使用偶数ID
帧（Frame）	流中的最小数据传输单元，每个帧属于某个特定的流	二进制格式；包含流ID标识；多种类型；大小固定

多路复用的工作流程如下：

(2) 客户端和服务器建立 TCP 连接。
(3) 客户端在连接上创建多个流，每个流对应一个请求。
(4) 客户端将请求消息分割为多个帧，并通过流发送给服务器。
(5) 服务器接收帧，根据流 ID 将帧重组为完整的请求。
(6) 服务器处理请求后，将响应消息分割为多个帧，并通过流发送给客户端。
(7) 客户端接收帧，根据流 ID 将帧重组为完整的响应。

图3 HTTP/2 多路复用示意图



3. 头部压缩

HTTP/1.1 每次请求都要发送完整的头部信息，即使内容重复（如 User-Agent、Cookie 等），造成带宽浪费。在高并发场景下，这些冗余数据会明显拖慢性能。

HTTP/2 通过 HPACK 压缩算法优化头部传输：

- 静态表：内置 61 个常用头部字段（如 GET 方法），直接用数字编号代替完整字段。
- 动态表：记录连接中已发送的头部，后续请求只需引用编号即可。

HTTP/3 升级为 QPACK 方案，在保持 HPACK 核心机制的同时，针对 QUIC 协议进行了优化：

- 允许头部/数据帧独立传输，避免丢包阻塞影响头部压缩效率。
- 优化动态表同步，支持连接迁移时头部保持压缩状态。

2.2 TCP协议的性能瓶颈

TCP（Transmission Control Protocol，传输控制协议）是互联网的核心协议之一，自 1974 年首次提出以来，它一直是可靠数据传输的基石。然而，随着网络技术的快速发展和应用场景的多样化，TCP 协议在现代网络环境中逐渐暴露出一些性能瓶颈，亟需面向未来的传输层协议来解决这些问题。

2.2.1 队头阻塞

HTTP/2 通过引入多路复用机制，允许在单个连接上并行传输多个数据流，成功解决了应用层的队头阻塞问题。然而，HTTP/2 仍然依赖于底层的 TCP 协议传输数据。

TCP 协议通过序列号机制确保数据的按序传递和处理，这一特性虽然保证了数据的可靠性，但也带来了潜在的队头阻塞问题。如果某个数据包未能按序到达或丢失，接收端会等待该数据包的重传，导致后续已到达的数据包无法被立即处理或发送。

在 TCP 协议中，队头阻塞现象可以出现在发送窗口和接收窗口两个不同的场景中。

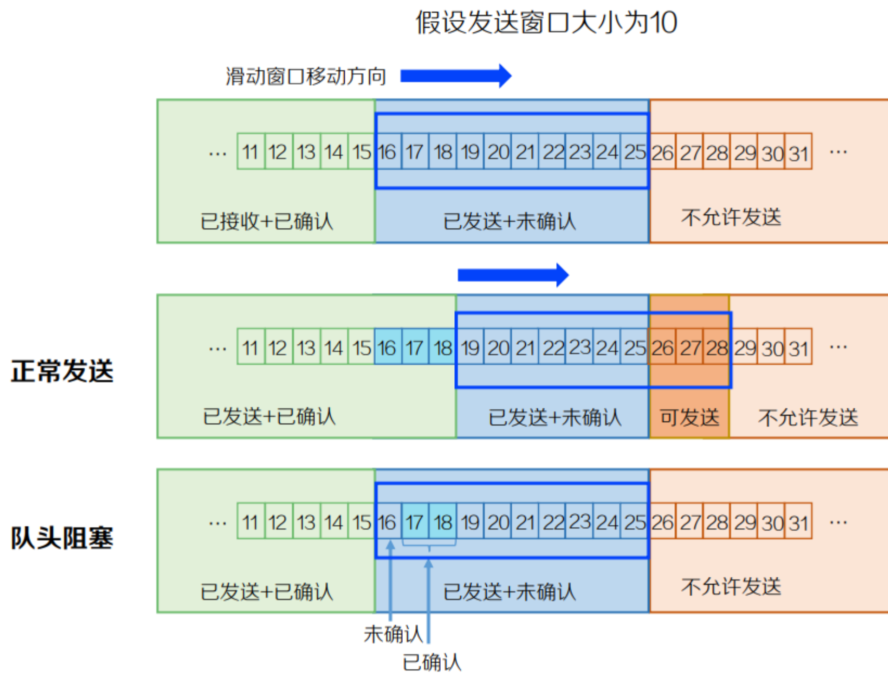
1. 发送窗口队头阻塞

发送窗口用于控制发送端允许发送的数据包范围。在发送端发送数据时，只有发送窗口内的数据包可以被发送，且需要依次收到接收端的确认（ACK），窗口才会向右滑动，允许更多的数据包进入窗口范围并被发送。如果窗口内的某些数据包未收到确认，窗口将保持不动，无法继续发送超出窗口范围的报文。

如图 4 所示，假设发送端已经成功发送了序号 1 到 15 的数据包，并收到 ACK 确认，当前窗口范围滑动到 16 到 25。这意味着序号 16 到 25 的数据包可以被发送。然而，由于滑动窗口协议要求窗口的滑动必须以窗口头部的数据包（即队头）收到 ACK 确认为前提，可能会出现以下情况：

- 如果从窗口头部开始的数据包依次收到 ACK 确认，则发送窗口可以顺利向右滑动。例如，当序号 16 到 18 的数据包相继收到 ACK 确认后，发送窗口将向右滑动 3 个位置。这使得序号 26 到 28 的数据包进入发送窗口范围，可以被发送到接收端。
- 如果序号 16 的报文未收到 ACK 确认，即使序号 16 之后的报文（如序号 17 和 18）已收到 ACK 确认，发送窗口依然无法滑动，导致无法发送新的数据包，形成队头阻塞现象。此时，发送端只能等待序号 16 的报文超时重传，并在成功收到 ACK 确认后，窗口才会向右滑动，解除阻塞状态。

图4 发送窗口队头阻塞



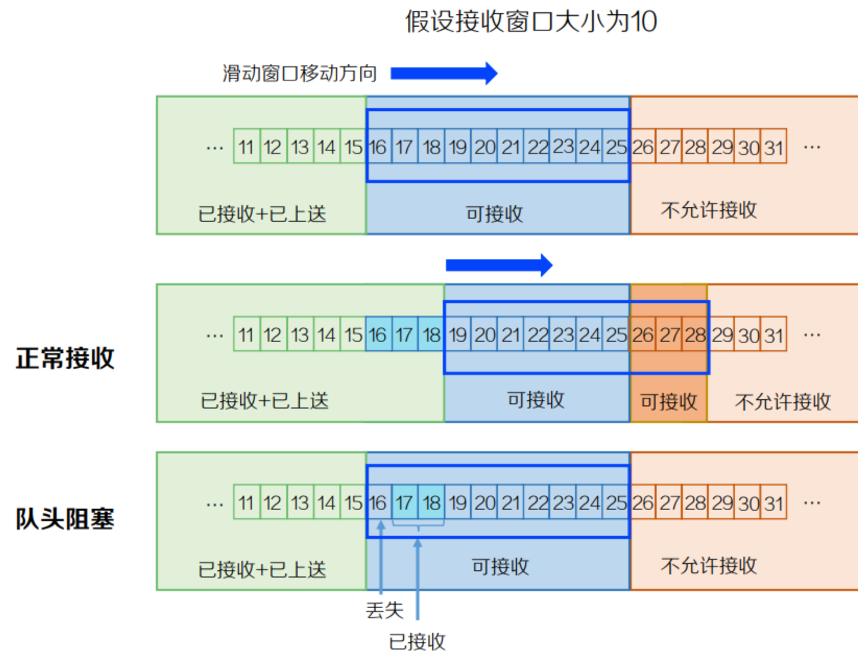
2. 接收窗口队头阻塞

接收窗口用于控制接收端可以接收的数据包范围。只有在接收窗口内的数据包可以被接收，并且只有当这些数据包按序到达并被上送到应用层后，接收窗口才会向右滑动，允许接收更多的数据包。如果窗口内的某些数据包未按序到达，窗口将保持不动，无法继续接收超出窗口范围的报文。

如图 5 所示，假设接收端已成功接收并上送了序号 1 到 15 的数据包，当前接收窗口范围为序号 16 到 25。这意味着这些序号的数据包可以被接收。然而，由于接收窗口的滑动必须以窗口头部的数据包被成功接收并上送为前提，可能会出现以下情况：

- 当接收端依次收到序号 16 到 18 的数据包，并且这些数据包是有序的，接收端可以将它们上送到应用层。此时，接收窗口将向右滑动 3 个位置，使得序号 26 到 28 的数据包进入接收窗口范围，可以被正常接收。
- 如果接收端先收到序号 16 之后的数据包，例如序号 17 和 18，由于序号 16 尚未接收，数据包的顺序不完整。这些乱序到达的数据包无法被上送到应用层，导致接收窗口无法滑动，从而形成队头阻塞现象。在这种情况下，接收端必须等待发送端重传序号 16 的数据包，只有成功接收并上送序号 16 的数据包后，接收窗口才得以滑动，继续接收新的数据包。

图5 接收窗口队头阻塞



2.2.2 高握手延迟

握手延迟是指在计算机网络中传输数据前，为建立连接而完成握手过程所需的时间，通常以往返时间（Round-Trip Time，RTT）作为衡量标准。作为确保通信双方能够高效发送和接收数据的关键步骤，握手过程的效率直接决定了连接建立的速度和整体性能。然而，在 TCP+TLS 握手过程中，由于涉及多次往返交互，RTT 的累积消耗较长，这在一定程度上限制了性能的进一步提升。

注意

目前，TLS 协议的主要版本包括 TLS 1.0、TLS 1.1、TLS 1.2 和 TLS 1.3。由于软硬件兼容性限制，当前与 TCP 结合的主流 TLS 版本仍为 TLS 1.2。以下握手延迟计算主要围绕 TLS 1.2 展开说明。如果采用 TLS 1.3，其握手过程将比 TLS 1.2 减少 1 个 RTT。

1. 首次连接

首次连接的总握手延迟=TCP 三次握手延迟+TLS 四次握手延迟=3RTT。

(1) TCP 三次握手

a. 客户端向服务器发送一个 SYN 报文，包含序列号和其他 TCP 选项。

b. 服务器收到 SYN 报文后，回复一个 SYN-ACK 报文，包含服务器的序列号和确认号。

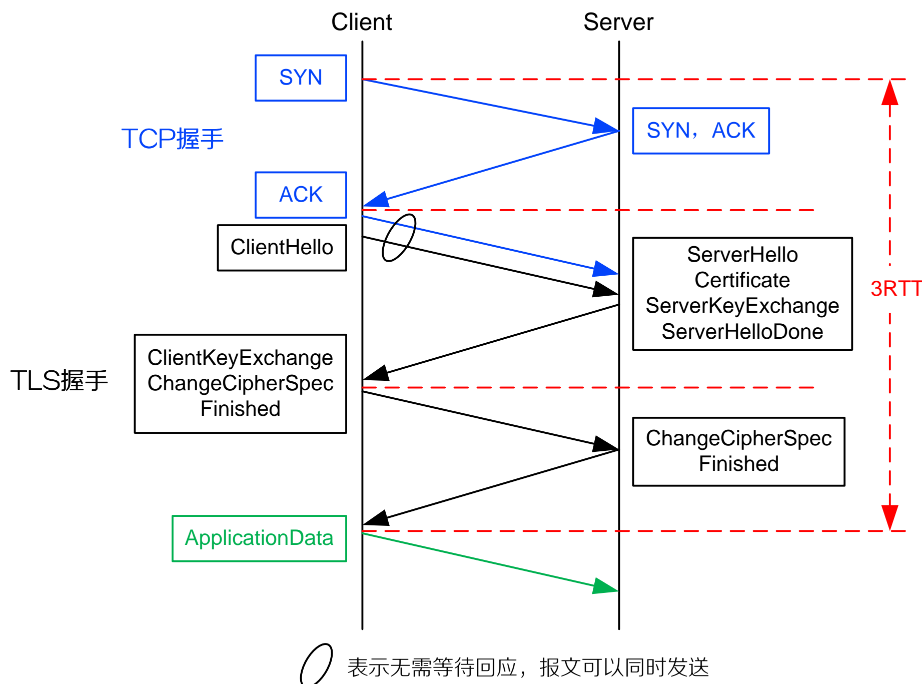
c. 客户端收到 SYN-ACK 报文后，发送一个 ACK 报文，确认连接建立。

(2) TLS 四次握手

a. 客户端向服务器发送ClientHello消息，包含支持的TLS版本、加密套件、随机数等信息。

b. 服务器收到 ClientHello 后，回复 ServerHello 消息，选择 TLS 版本、加密套件和随机数，并发送 Certificate 消息（包含服务器的公钥证书）和 ServerKeyExchange 消息（如果需要），最后发送 ServerHelloDone 消息，表示握手的第一阶段完成。
c. 客户端生成预主密钥（Pre-Master Secret），通过 ClientKeyExchange 消息发送给服务器，随后发送 ChangeCipherSpec 和 Finished 消息，表示后续通信将使用协商参数进行加解密。
d. 服务器收到 ClientKeyExchange 后，解密预主密钥并生成主密钥（Master Secret），同样发送 ChangeCipherSpec 和 Finished 消息，表示后续通信将使用协商参数进行加解密，握手结束。

图6 首次连接 TCP+TLS 握手过程



2. 会话复用（恢复连接）

在 TCP+TLS 会话保留恢复连接时，如果客户端和服务器支持会话复用（Session Resumption），可以通过复用之前的会话信息来减少握手过程的消耗，从而将连接建立的时间从 3RTT 降低到 2RTT。

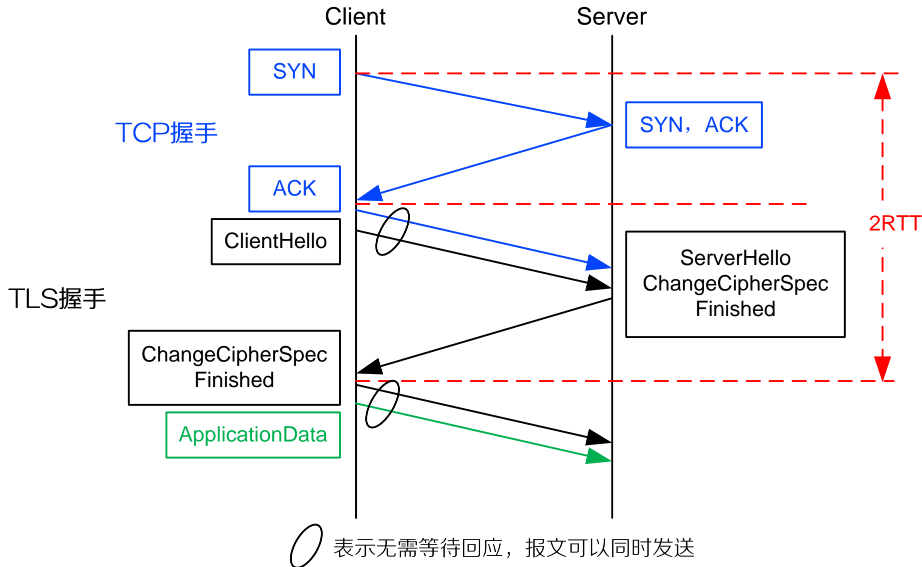
(1) TCP 三次握手

同首次连接。

(2) TLS 握手

a. 客户端向服务器发送 ClientHello 消息，包含支持的 TLS 版本、加密套件、随机数、会话 ID 等信息。
b. 服务器收到 ClientHello 后，检查会话 ID。如果会话有效，服务器直接复用之前的会话信息，生成新的主密钥（Master Secret），并回复 ServerHello 消息，包含选择的 TLS 版本、加密套件和随机数。同时回复 ChangeCipherSpec 和 Finished 消息，表示后续通信将使用协商参数进行加解密。

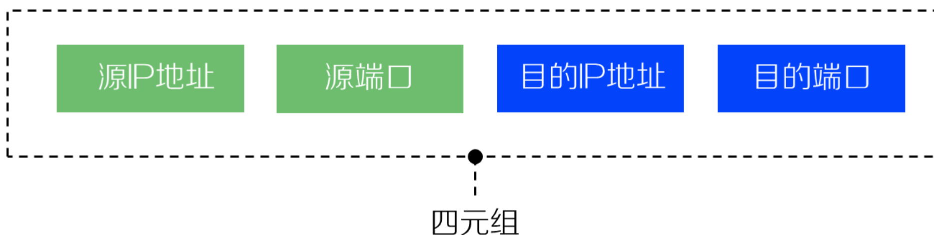
c. 客户端收到 ServerHello 后，验证服务器的 Finished 消息，确认会话恢复成功。客户端发送 ChangeCipherSpec 和 Finished 消息，表示后续通信将使用协商参数进行加解密，握手结束。



2.2.3 连接迁移的缺失

TCP 连接依赖于四元组（源 IP、目的 IP、源端口和目的端口）来唯一标识。然而，当四元组中的任一信息发生变化时，例如客户端从移动网络（4G/5G）切换到 Wi-Fi 导致 IP 地址改变，TCP 连接将中断并需要重新建立。这一过程涉及 TCP 三次握手和 TLS 四次握手的延迟，以及 TCP 慢启动的降速机制，导致网络切换时出现卡顿，影响用户体验的流畅性。

图7 四元组信息



2.2.4 加密与安全的局限性

TCP 协议本身缺乏加密功能，其安全性依赖于上层的 TLS 协议。这种分层设计虽然灵活，但也带来了一些挑战：

- 延迟增加：TLS 握手过程会引入额外的 RTT，导致连接建立延迟。尽管 TLS 1.3 通过简化握手流程提升了效率和安全性，但部分旧设备或软件可能不支持该版本，引发兼容性问题。
- 复杂性高：TLS 协议涉及多种加密算法、密钥交换机制和证书验证流程，其复杂性增加了配置错误或实现漏洞的风险。
- 效率较低：由于加密和传输逻辑分离，实现起来可能更为复杂且效率较低。此外，TLS 会引

入额外的协议头和数据填充，导致数据包增大，可能影响网络传输效率。

- 安全性局限：TCP 加 TLS 通常只加密客户端和服务器之间的通信，中间节点(如 CDN、代理)可能能够解密和查看数据，无法实现真正的端到端加密。

2.2.5 拥塞控制的保守性

TCP 的拥塞控制机制，如慢启动和拥塞避免，旨在通过降低发送速率来缓解网络拥塞。然而，这种保守的策略在动态网络环境中，尤其是实时应用（如在线游戏或实时视频流）中，可能导致吞吐量不足，影响用户体验。以下是 TCP 拥塞控制的一些主要局限性：

- 对高带宽和高延迟网络表现不佳：慢启动阶段从较小的拥塞窗口开始，逐步增加发送速率，而拥塞避免阶段的线性增长速度较慢。这意味着在高带宽高延迟网络中，TCP 可能需要较长时间才能达到最大吞吐量，导致带宽利用率低下。
- 对非拥塞丢包的过度敏感：TCP 将丢包视为拥塞的信号，并大幅减小拥塞窗口。然而，在无线网络或链路质量较差的环境中，丢包可能是由其他因素(如无线信号干扰或比特错误)引起的，而非拥塞。这会导致 TCP 错误地降低发送速率，进而影响性能。
- 对短流和突发流不友好：TCP 拥塞控制机制（如慢启动）主要针对长连接优化，而短流（如 HTTP 请求）可能在完成传输之前无法充分利用带宽。突发流量（如视频流或文件传输）可能导致拥塞窗口频繁调整，影响传输效率。
- 对应用需求的适应性不足：不同的应用对网络性能的需求不同(如低延迟、高吞吐量、公平性等)，修改 TCP 拥塞控制机制以满足所有需求非常困难。

这些局限性表明，尽管 TCP 的拥塞控制机制在避免网络拥塞和确保公平性方面表现出色，但在某些条件下仍需优化以提高网络性能和用户体验。

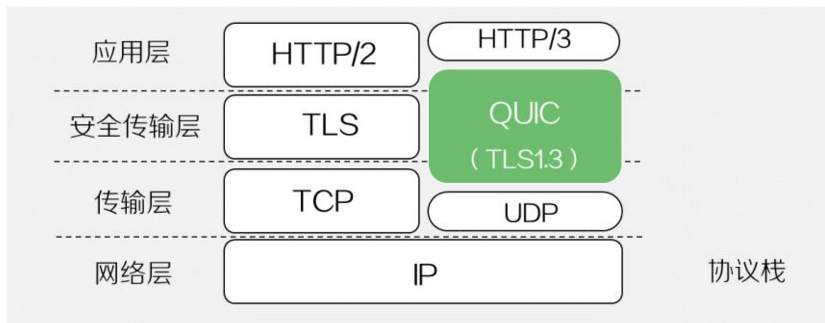
3 QUIC 协议技术实现

3.1 QUIC协议架构

3.1.1 QUIC 协议栈

如图 8 所示，QUIC（Quick UDP Internet Connections）是一种基于 UDP 的传输层协议。与 TCP 协议在传输层之上独立运行 TLS 和 HTTP/2 的传统架构不同，QUIC 通过创新设计，将 TLS 1.3 深度集成到其协议栈中，作为内置的安全机制，同时融合了部分应用层功能(如流管理和多路复用)。这种设计不仅减少了协议分层，还优化了整体结构，使得 QUIC 在简化协议栈的同时，显著提升了连接效率和性能，为现代网络通信提供了更加高效和安全的解决方案。

图8 QUIC 协议栈示意图



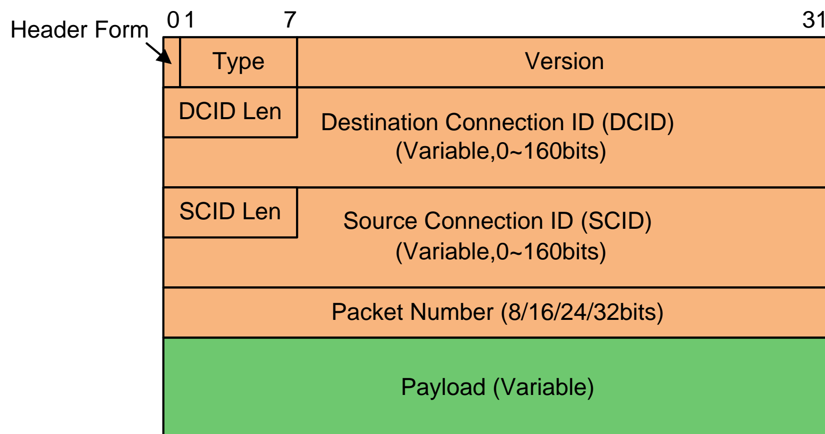
3.1.2 QUIC 协议报文格式

在 QUIC 协议中，报文类型主要分为两大类：长头部报文（Long Header Packet）和短头部报文（Short Header Packet）。

1. 长头部报文格式

长头部报文用于连接建立、版本协商和初始通信等场景，其报文格式如图 9 所示。

图9 长头部报文格式



长头部报文的各部分字段解释如下：

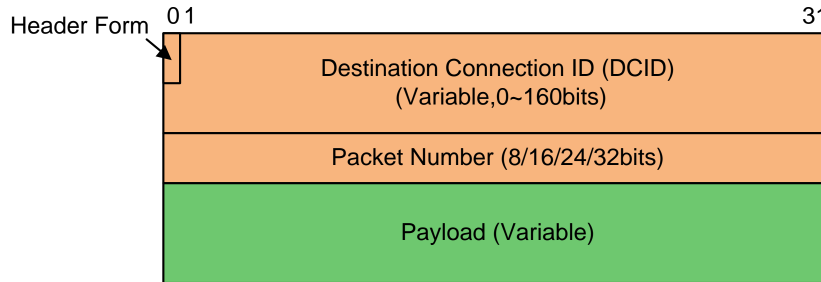
- Header Form：固定为 1，表示长头部报文。
- Type：报文类型，取值包括：
  - Initial 报文（0x00）：初始握手报文。
  - 0-RTT 报文（0x01）：0-RTT 数据报文，在连接恢复时发送早期应用数据。
  - Handshake 报文（0x02）：握手阶段报文。
  - Retry 报文（0x03）：服务端拒绝 Initial 报文时发送的应答报文。
- Version：QUIC 协议版本号。
- DCID Len：目标连接 ID（Destination Connection ID）的长度。
- Destination Connection ID：目标连接 ID，标识接收方。

- SCID Len：源连接 ID（Source Connection ID）的长度。
- Source Connection ID：源连接 ID，标识发送方。
- Packet Number：报文编号，用于排序和重传。
- Payload：有效载荷，包含 QUIC 帧，例如传输数据的 STREAM 帧、确认数据接收的 ACK 帧等。

2. 短头部报文格式

短头部报文，即 1-RTT 报文，用于建立连接后的高效通信，其报文格式如图 10 所示。

图10 短头部报文格式



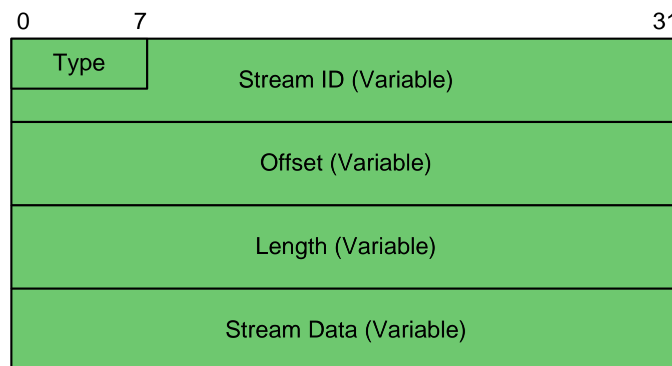
短头部报文的各部分字段解释如下：

- Header Form：固定为 0，表示短头部报文。
- Destination Connection ID：目标连接 ID，标识接收方。
- Packet Number：报文编号，用于排序和重传。
- Payload：有效载荷，包含 QUIC 帧（如 STREAM 帧、ACK 帧等）。

3. STREAM 帧格式

帧是 QUIC 报文的有效载荷，用于传输具体的数据和控制信息。每个 QUIC 报文可以包含一个或多个帧。帧的类型有很多种，每种帧的头部不尽相同，以 STREAM 帧为例，其帧格式如图 11 所示。

图11 STREAM 帧格式



STREAM 帧的各部分字段解释如下：

- Type：帧类型，STREAM 帧的值为 0x08。
- Stream ID：流标识符，标识数据所属流。
- Offset：数据在流中的起始字节偏移量，单位为字节。

- Length：数据长度。
- Stream Data：实际的应用层数据。

3.2 QUIC协议的性能优化

3.2.1 显著降低握手延迟

相比传统的 TCP+TLS，QUIC 通过基于 UDP 的设计、集成 TLS 加密等机制，可以在首次和重复连接中更快地建立安全连接，减少握手延迟。

1. 首次连接

当客户端首次与服务器建立 QUIC 连接时，消耗 1RTT，其完整的握手流程如下：

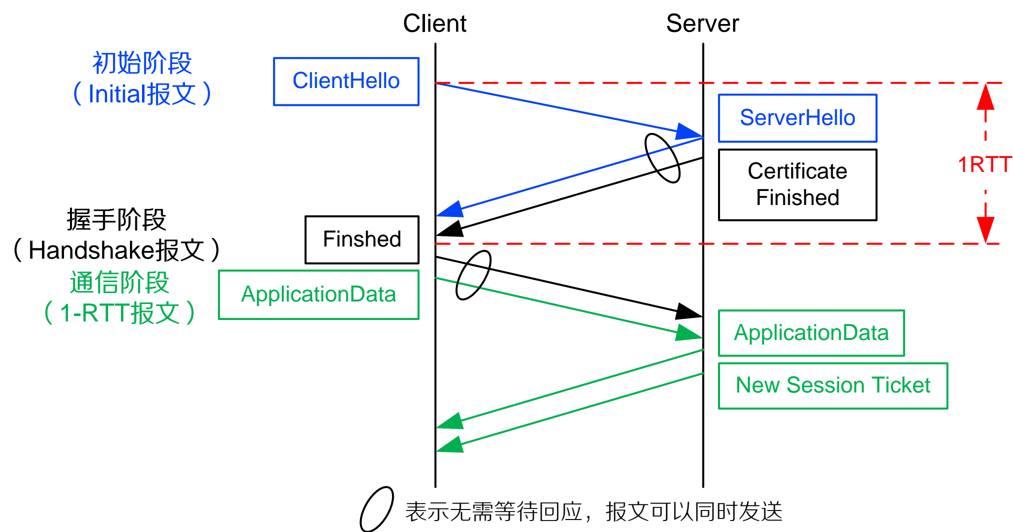
(1) 客户端发送 Initial 包，其中包含完整的 Client Hello 消息。Client Hello 消息中包含 QUIC 和 TLS 版本号、加密套件、客户端生成的随机数和公钥等信息，用于协商加密参数。

(2) 服务器收到客户端发送的 Initial 包后，会进行以下处理：
   - 验证版本号，选择支持的加密套件。
   - 使用客户端随机数和服务器私钥生成共享密钥，并派生出会话密钥，用于加密后续通信。
   - 回复 Server Hello 消息，包含服务器选择的协议版本、加密套件和服务器公钥等信息。
   - 将服务器证书和 Finished 消息封装在 Handshake 包中，并发送给客户端。

(3) 客户端收到 Server Hello 消息、证书和 Finished 消息后，会进行如下处理：
   - 客户端检查服务器选择的 QUIC 和 TLS 版本，确保与客户端支持的版本兼容。
   - 验证服务器证书，确保服务器身份的合法性。
   - 使用服务器随机数和客户端私钥生成共享密钥，并派生出会话密钥，用于加密后续通信。
   - 验证 Finished 消息，确保握手过程的完整性。
   - 生成自己的 Finished 消息，封装在 Handshake 包中发送给服务器。

(4) 服务器收到客户端的 Finished 消息后，会进行如下处理：
   - 验证 Finished 消息，确保握手过程的完整性。
   - 如果服务器支持会话恢复，则它会在握手完成后发送 New Session Ticket 消息，其中包含会话票证（Session Ticket），以便客户端在后续连接中快速恢复会话。

图12 QUIC 首次连接



2. 会话恢复

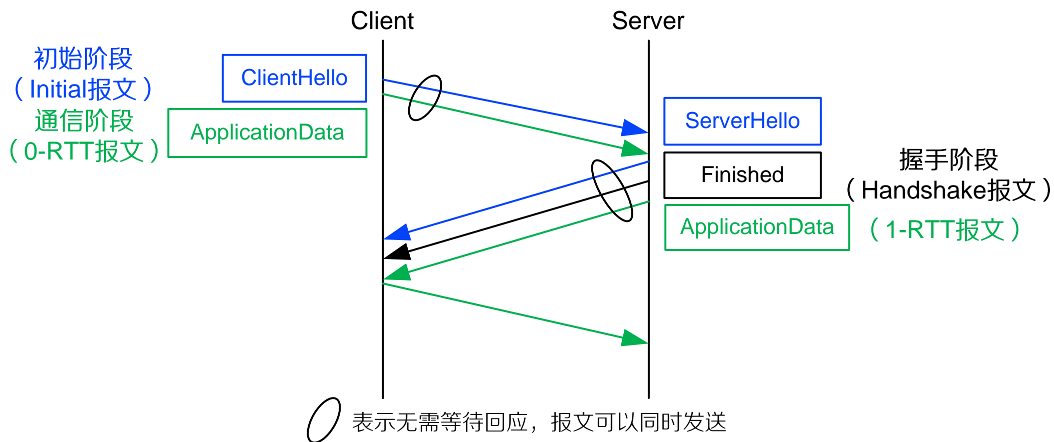
QUIC 协议支持 0-RTT 传输，即允许客户端在恢复连接时，在第一次握手报文中直接发送应用数据，而无需等待完整的握手过程完成。0-RTT 传输流程如下：

(1) 客户端在重连同一服务器时，会进行如下操作：
- 发送 Client Hello 消息给服务器，其中包含缓存的会话票证和客户端生成的新的随机数等信息。
- 基于首次连接的会话密钥和新的随机数派生出 0-RTT 密钥，使用 0-RTT 密钥加密应用请求并发送给服务器。

(2) 服务器收到 Client Hello 消息和 0-RTT 数据包后，会进行如下操作：
- 验证 Client Hello 消息中的会话票证：
  - 如果会话已经过期，服务器会发送 Server Hello 消息、证书和 Finished 消息，要求与客户端重新建立连接。过程同首次连接。
  - 如果会话没有过期，服务器会发送 Server Hello 消息（包含客户端新的随机数）和 Finished 消息，确认恢复会话。
- 基于客户端的随机数和首次连接的会话密钥派生出 0-RTT 密钥，使用 0-RTT 密钥解密应用请求。
- 基于客户端的随机数、服务器的随机数和首次连接的会话密钥派生出新的会话密钥，使用新的会话密钥加密应答数据，并发送给客户端。

(3) 客户端收到 Server Hello 消息和 Finished 消息后，会进行如下处理：
- 验证 Server Hello 消息和 Finished 消息，确认会话恢复成功。
- 基于服务器随机数、客户端的随机数和首次连接的会话密钥派生出相同的会话密钥，使用该会话密钥解密应答数据。
- 后续客户端和服务器均使用新的会话密钥加密通信。

图13 QUIC 会话恢复



3.2.2 增强型可靠传输

在 TCP 协议中，为了确保可靠传输，TCP 协议使用字节序列号和确认号来确认消息的有序到达。如果数据包在传输过程中丢失并需要重传，其编号保持不变。

QUIC 使用包编号（Packet Number）来标识每个数据包，而流偏移量（Stream Offset）用于标识流中的数据位置：

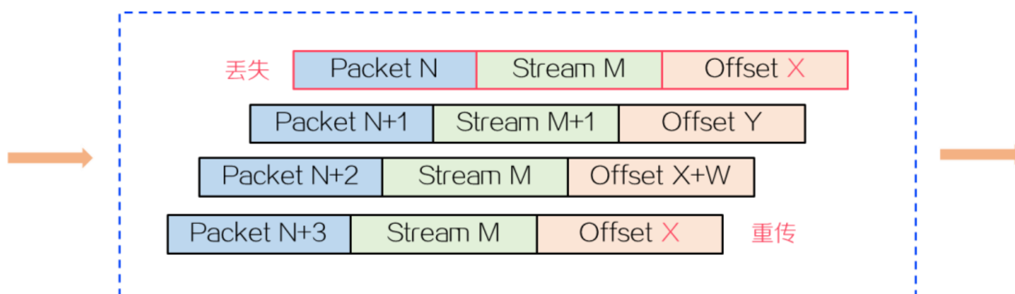
- 包编号是单调递增的，即使重传也不会重复使用相同的编号，这有助于避免 TCP 中“重传歧义”问题（即无法区分应答报文是对原始包还是重传包的确认）。
- 流偏移量表示帧在整个流中起始位置的偏移，确保即使包编号不连续，仍然可以根据其偏移位置按正确顺序组装和处理。

包编号和流偏移量的工作机制如图 14 所示：

(1) Packet N 到 Packet N+2 通过同一 QUIC 连接发送，其中 Packet N 和 Packet N+2 属于同一数据流。当 Packet N 丢失时，发送端重传该数据包，编号变更为 Packet N+3，Offset 不变。

(2) 接收端收到 Packet N+2、 Packet N+3 数据包时，只依据 Stream ID 无法确认组装顺序，此时查看其偏移值，发现 Packet N+2 的偏移量比 Packet N+3 大，则组装时 Packet N+2 位于 Packet N+3 之后。

图14 QUIC 可靠传输

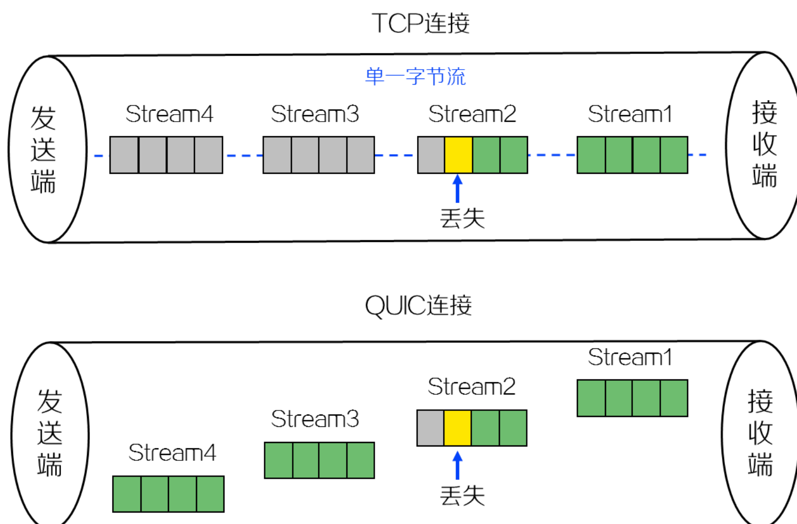


3.2.3 高效的传输层多路复用

TCP 协议利用序列号确保数据按顺序传递和处理。在单一字节流中实现多个数据流传输时，如果某个数据包未按序到达，TCP 会等待该数据包的重传，导致后续数据包无法立即被处理或发送，进而引发队头阻塞问题。因此，在高丢包率、高延迟或带宽受限的网络环境中，使用 TCP 协议可能导致延迟增加和吞吐量下降，从而影响网络应用的性能和用户体验。

QUIC 允许多个数据流在单个连接中并行传输，每个数据流拥有独立的顺序和确认机制。因此，即使一个流出现阻塞，也不会影响其他流的传输。这种特性大大提高了数据传输的效率和可靠性。

图15 TCP 和 QUIC 数据流传输对比



3.2.4 灵活的流量控制

在多路复用的背景下，TCP 连接通过单一字节流进行数据传输，无法区分和独立控制各个数据流，因此只能实现连接级别的流量控制。相比之下，QUIC 中的数据流彼此独立，从而支持流级（Stream）和连接级（Connection）的流量控制。

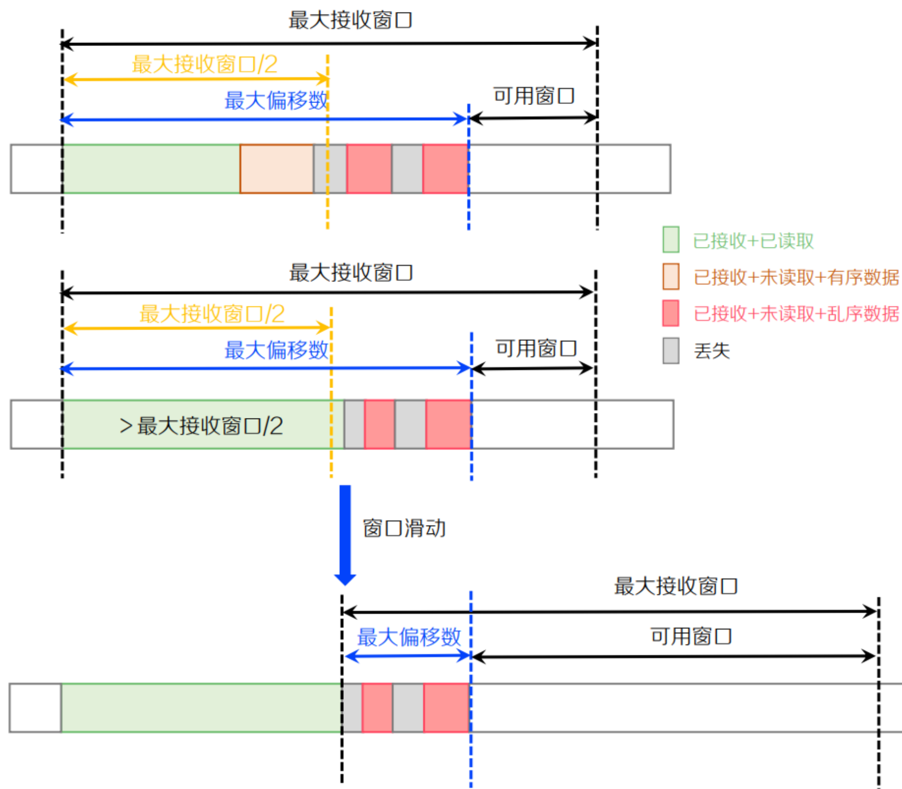
1. 流级流量控制

QUIC 支持多路复用，允许在单个连接中同时传输多个独立的流。每个流都有自己的流量控制窗口，独立于其他流。接收方可以为每个流设置单独的窗口大小，控制每个流的数据传输速率。这种设计避免了单一流占用过多资源，影响其他流的传输。

流级流量控制的机制如下：

- 接收窗口的初始大小为最大接收窗口。数据接收过程中，可用窗口逐渐减小，其计算公式为：可用窗口=最大接收窗口-最大偏移数。其中，最大偏移数表示理论上应该接收到的数据总量。
- 当接收端已接收并读取的数据量超过最大接收窗口的一半时，触发窗口滑动。窗口滑动后，最大接收窗口的大小保持不变，但其左边界移动到已接收并读取的数据末端。可用窗口的左边界也移动到已读取数据的末端，右边界与最大接收窗口保持一致。
- 当接收窗口滑动后，接收端会向发送方发送窗口更新帧，通知发送方新的窗口边界。发送方收到窗口更新帧后，即使中途存在丢包，也会根据新的窗口边界调整发送窗口，确保数据发送与接收端的处理能力匹配。

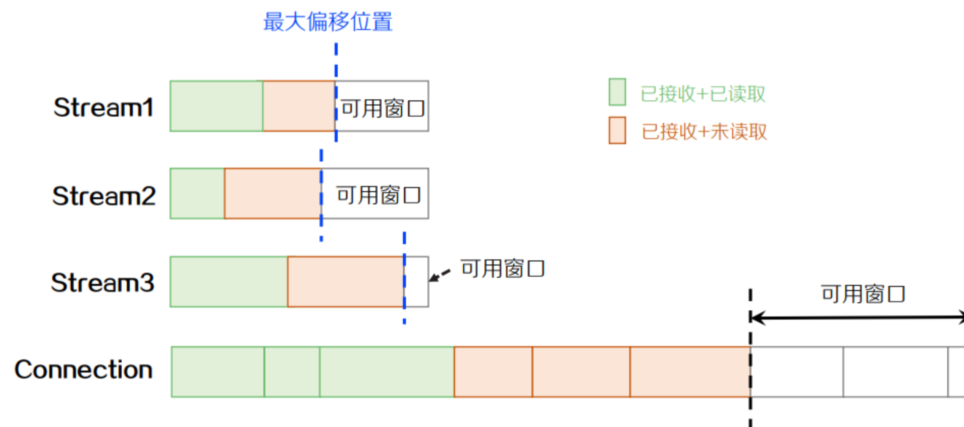
图16 流级流量控制



2. 连接级流量控制

除了基于流的流量控制，QUIC 还实现了基于整个连接的流量控制。接收方可以为整个连接设置一个总的流量控制窗口，限制所有流的总数据量。这种机制确保整个连接不会超出接收方的处理能力。Connection 级别的可用窗口等于该连接上所有 Stream 可用窗口的总和。

图17 连接级流量控制



3.2.5 智能化的拥塞控制

1. 可插拔的拥塞控制算法

TCP 的实现通常依赖于操作系统内核，拥塞控制算法（如 Cubic、Reno 等）是内核的一部分。因此，修改或升级 TCP 的拥塞控制算法需要修改内核代码，并重新编译和部署操作系统。由于内核升级涉及系统稳定性、兼容性等问题，TCP 的拥塞控制算法更新通常需要较长的周期。

QUIC 支持可插拔的拥塞控制算法，允许开发者动态地选择、替换或实现不同的拥塞控制算法，而无需修改协议的核心实现或依赖操作系统的支持。可插拔机制的优势如下：

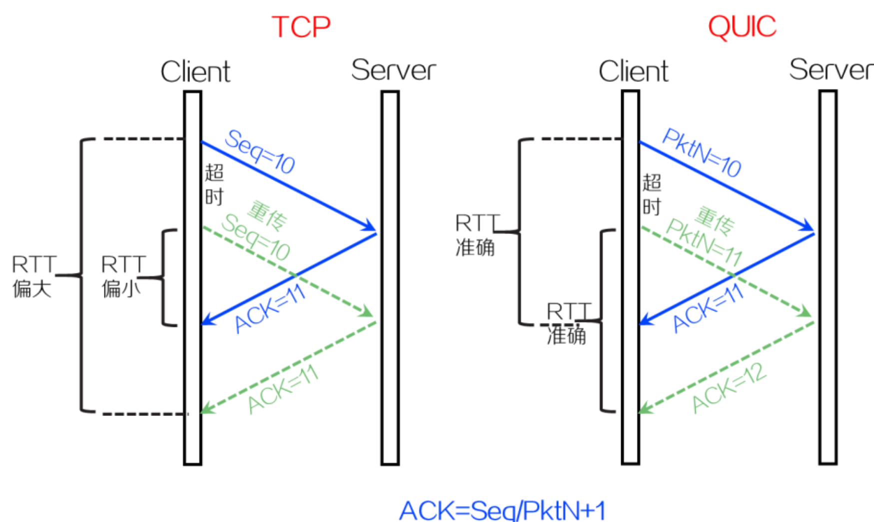
- 部署和升级便捷：由于 QUIC 的拥塞控制算法是用户态代码的一部分，服务端只需修改配置或重新加载服务即可实现算法的切换。
- 灵活的算法选择：开发者可以根据网络环境或业务需求快速切换不同的算法（如 Cubic、BBR、Reno 等），甚至在同一服务中为不同用户或连接使用不同的算法。
- 支持定制化需求：开发者可以根据具体需求定制拥塞控制算法，例如针对特定网络环境（如移动网络、卫星网络）或特定应用场景（如低延迟、高吞吐量）进行优化。
- 跨平台一致性：由于 QUIC 在用户态实现，其行为在不同操作系统（如 Linux、Windows、macOS）上是一致的，避免了 TCP 在不同平台上可能存在的实现差异。

2. 更精确的 RTT 测量

RTT 是反映网络延迟的关键指标，它能够动态地反映当前网络的拥塞状态。准确的 RTT 测量可以帮助拥塞控制算法及时了解网络情况，以便做出合适的调整。图 18 显示了 TCP 和 QUIC 在 RTT 计算方面的差异：

- TCP 连接：由于初始数据包和重传包使用相同的序列号，无法区分是初始传输还是重传。这可能导致存在数据重传的情况时，RTT 计算出现偏差，结果可能偏小或偏大，从而影响拥塞控制的效果。
- QUIC 连接：采用递增的包编号来标识每个数据包，初始包和重传包拥有不同的编号，这种设计避免了对重传包的误识别，从而保证了 RTT 计算的准确性。

图18 RTT 计算



3.2.6 支持无缝连接迁移

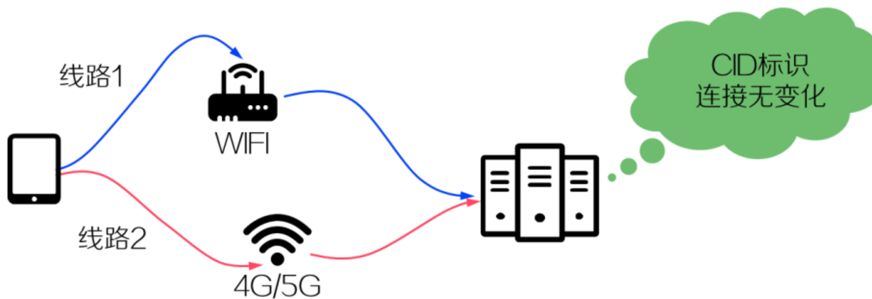
QUIC 基于 UDP 协议，而 UDP 属于无连接传输协议，没有连接的概念。为此，QUIC 利用终端生成的可变长度的随机数作为连接标识符（Connection ID，简称为 CID）来区分不同连接。

(1) 在建立连接阶段，客户端和服务端会各自生成一组 CID 并发布给对端，其中每个 CID 都可以标识本次连接，但并非所有 CID 都会同时使用。通常，客户端和服务端会选择一个主 CID 用于当前通信，其他 CID 作为备用。

(2) 在数据传输阶段，每个 QUIC 数据包的头部会包含一个目的 CID（DCID），用于标识数据包的目标连接。对端通过 DCID 识别数据包所属的连接，而不是依赖源 IP 地址和端口。

(3) 当网络线路发生变化时，客户端或服务端可以选择从已协商的 CID 组中选择一个新的 CID 作为主 CID，以区分不同的网络线路。CID 的切换可以增强隐私性，防止追踪，因为攻击者无法轻易关联新旧 CID。由于双方已知对端的所有 CID，且同一组中的 CID 均标识同一连接，因此 CID 切换后仍可以无缝复用现有连接。

图19 QUIC 连接迁移



4 TCP 与 QUIC 实现机制对比总结

如表 2 所示，QUIC 在 TCP 的基础上进行了多项改进，可以提供比 TCP 更快速、可靠和安全的互联网连接。但需要注意的是，尽管 TCP 在高丢包率、高延迟或带宽受限的网络下表现不佳，但其在稳定、高带宽的环境中仍表现优异。同时，鉴于其广泛的应用，TCP 在未来一段时间内仍会作为主流的传输层协议存在。根据具体的场景和网络条件，选择合适的协议才能获得最佳的性能和用户体验。

表2 QUIC 与 TCP 性能对比

协议 特征	TCP	QUIC
握手延迟	• 首次连接：3RTT • 会话恢复：2RTT	• 首次连接：1RTT • 会话恢复：0RTT
多路复用	多个数据流之间存在顺序依赖，中间流丢包会导致队头阻塞问题	基于UDP实现，多个流间不存在顺序依赖，中间流丢包不会导致队头阻塞
可靠传输	使用字节序号和确认号来确认消息的有序到达。如果数据包在传输过程中丢失并需要重传，其编号保持不变	使用递增的包编号和流偏移量来确保数据传输的可靠性和有序性
流量控制	单一字节流，无法区分不同流，仅支持进行连接级流量控制	单一连接中数据流之间相互独立，可同时进行流级和连接级别的流量控制
拥塞控制	• 依赖于内核和操作系统的实现，升级周期长，协议僵化 • 内核修改拥塞控制算法对所有应用连接生效 • 丢包情况下，拥塞控制算法计算 RTT 存在误差	• 不需要操作系统、内核的支持，在应用程序层面即可实现，便于部署、易升级，支持产品快速迭代 • 可以为单个应用程序的不同连接配置不同的拥塞控制算法，以实现不同应用场景的深度定制 • 拥塞控制算法计算 RTT 更准确
连接迁移	依赖于四元组标识连接，用户切换网络往往意味着需要进行TCP重连	采用CID作为连接标识，支持在用户切换网络时复用原连接，可以帮助用户进行平滑的网络切换、减少用户在接收视频流时的卡顿

5 典型应用

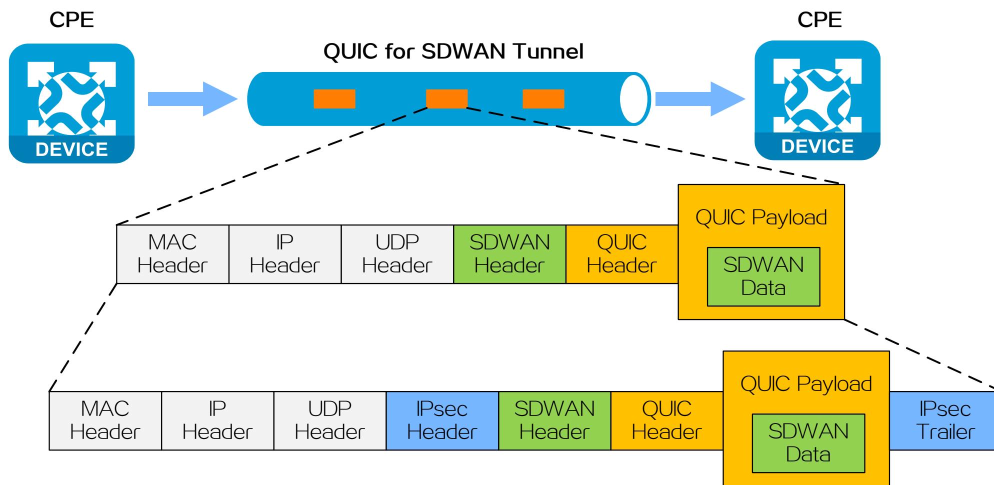
5.1 QUIC在SDWAN网络中的应用

如图 20 所示，在 SDWAN 隧道的两端 CPE 设备上集成 QUIC 协议，SDWAN 能够为用户提供一个更加智能化、灵活化和性能优越的网络服务解决方案：

- 用户体验无缝升级：用户侧的终端设备（如电脑、手机、平板等）不需要进行硬件升级或更换。所有的改进和功能增强都在网络基础设施层面进行，实现用户无感知的升级。
- 提高传输效率：通过 QUIC 协议的集成，显著减少了数据传输的时延，有效降低了网络卡顿率。特别是在高丢包率、高延迟或带宽受限的弱网环境中，QUIC 提供了更稳定的性能，确保数据传输的连贯性和速度。

- 灵活的业务质量保证：根据不同的业务需求，动态配置 QUIC 的拥塞控制算法。例如，可以为视频会议选择低延迟拥塞控制算法，而为大文件传输选择高吞吐量的算法，以便为用户提供个性化、灵活的业务质量保证。

图20 QUIC 在 SDWAN 网络中的应用



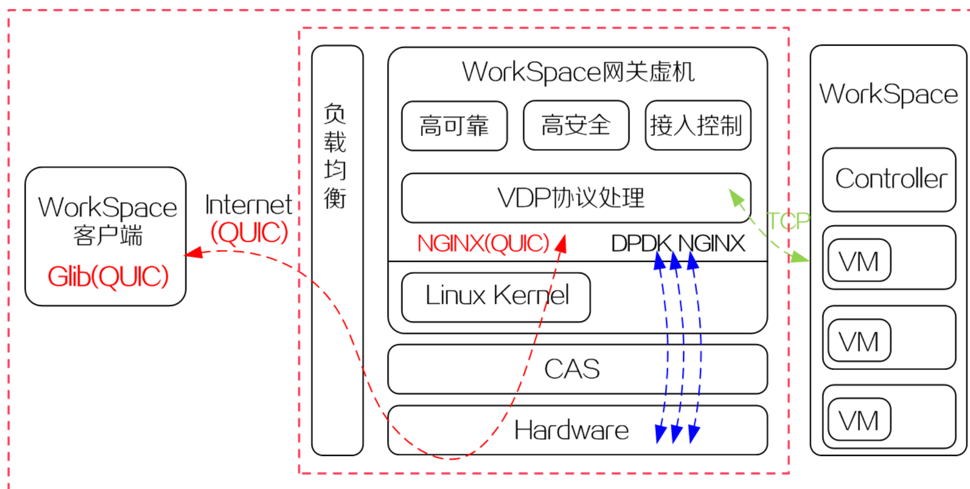
5.2 QUIC在VDI网关上的应用

在网关侧的 NGINX 中和 WorkSpace 客户端的 Glib 库中集成 QUIC，将客户端和网关之间的传输层协议从 TCP 替换为 QUIC。

- WorkSpace 客户端通过集成到 Glib 库中的 QUIC 携带 HTTP 请求。
- 负载均衡安全设备将 QUIC 请求转发到基于 NGINX 开发的 WorkSpace 网关虚机上。
- WorkSpace 网关虚机将 QUIC 请求代理成 TCP 请求，发送给业务模块。

在 VDI 网关上应用 QUIC 后，可以提升云桌面在弱网环境下的协议连接稳定性，在客户端连接发生迁移时，不断链继续服务。同时，利用 QUIC 灵活的拥塞控制，通过针对不同业务定制算法来提高公网环境下客户端到网关侧的传输性能，提升用户的体验。

图21 QUIC 在 VDI 网关上的应用



6 参考文献

- RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport
- RFC 9001: Using TLS to Secure QUIC
- RFC 9002: QUIC Loss Detection and Congestion Control