The QUIC Transport Protocol: Design and Internet-Scale Deployment

Adam Langley, Alistair Riddoch, Alyssa Wilk, et al.

SIGCOMM 17

授课教师：赵功名

中国科大计算机学院

2025年秋·高级计算机网络

QUIC 论文推荐：配套视频讲解（B 站）

- 为帮助大家更高效理解 QUIC 的核心设计与论文主线，推荐观看以下视频讲解（建议在阅读论文前/后各看一遍）：

[https://www.bilibili.com/video/BV1Mg411s7mP/]

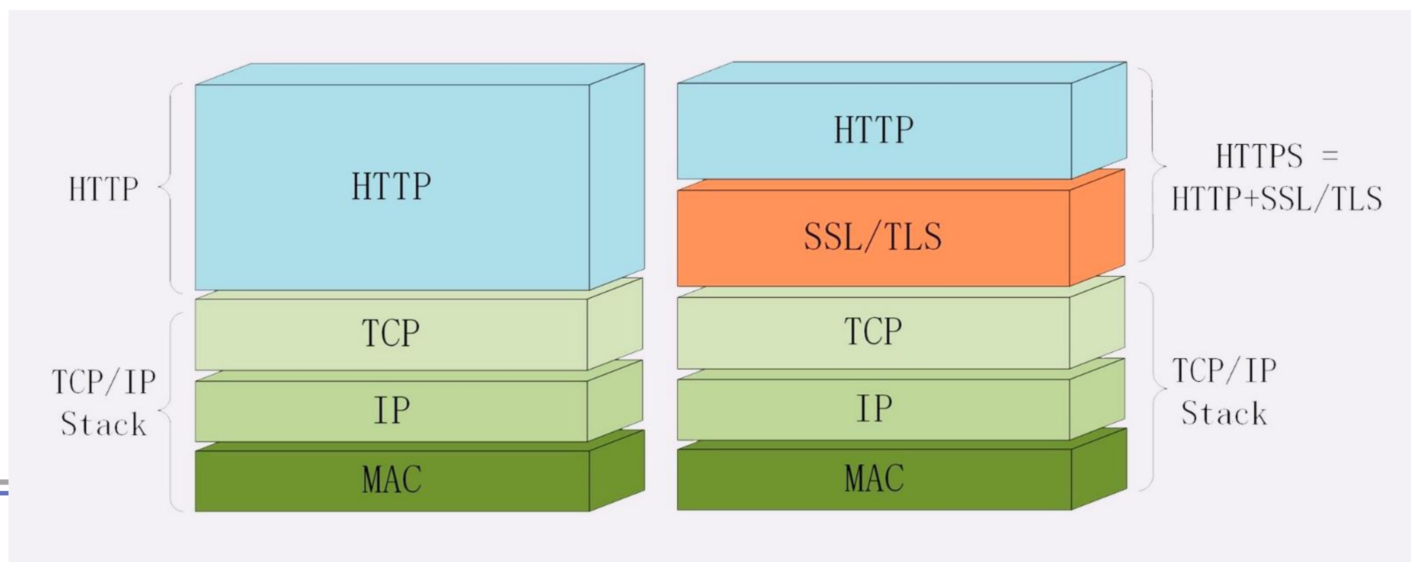
- 建议观看重点：
  - QUIC 的设计动机：为何要在 UDP 上重做传输层能力
  - 0-RTT / 1-RTT 握手与连接迁移（Connection Migration）
  - 多路复用与避免队头阻塞（HOL Blocking）
  - 与 TCP + TLS / HTTP/2 的关键差异与性能收益

Outline

1. Introduction
2. QUIC DESIGN AND IMPLEMENTATION
3. EXPERIMENTATION FRAMEWORK
4. INTERNET-SCALE DEPLOYMENT
5. QUIC PERFORMANCE
6. EXPERIMENTS AND EXPERIENCES

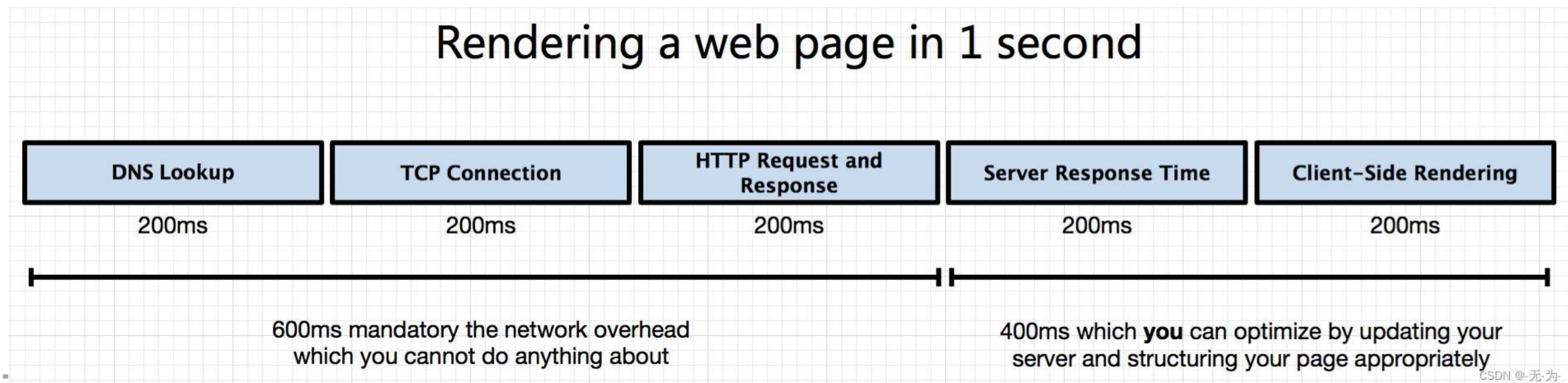
1. Introduction

- QUIC 简介
  - 一个“从零设计”的新传输层协议，目标是提升 HTTPS 性能并支持快速部署与持续演进
  - 在功能上替代传统 HTTPS 栈里大量组件：HTTP/2、TLS、以及底层 TCP（从应用视角
  - 设计选择：user-space 实现 + UDP 作为承载，让协议更新跟着应用迭代走
  - 核心关键词：低延迟、加密、复用(multiplexing)、可部署性



1. Introduction

- Web 业务越来越“怕延迟”
  - Web 从“页面浏览”变成“应用平台”，出现大量强延迟敏感服务（搜索、视频、交互式应用）
  - 尾延迟(tail latency) 仍然是规模化服务的关键障碍之一
  - 设计选择：user-space 实现 + UDP 作为承载，让协议更新跟着应用迭代走
  - 结论：传输层的“每一次多余 RTT”都会被放大为用户体验问题（为后面握手/HoL铺垫）



1. Introduction

- HTTPS 普及带来额外开销：安全与性能的矛盾
  - 互联网正从不加密快速迁移到加密流量，HTTPS 占比快速上升尾延迟(tail latency) 仍然是规模化服务的关键障碍之一
  - 加密是必然趋势，但它会带来额外握手与协议栈复杂度
  - 论文用 Google 前端服务的数据说明这一趋势：安全流量增长很快（现在更需要 QUIC 的证据)

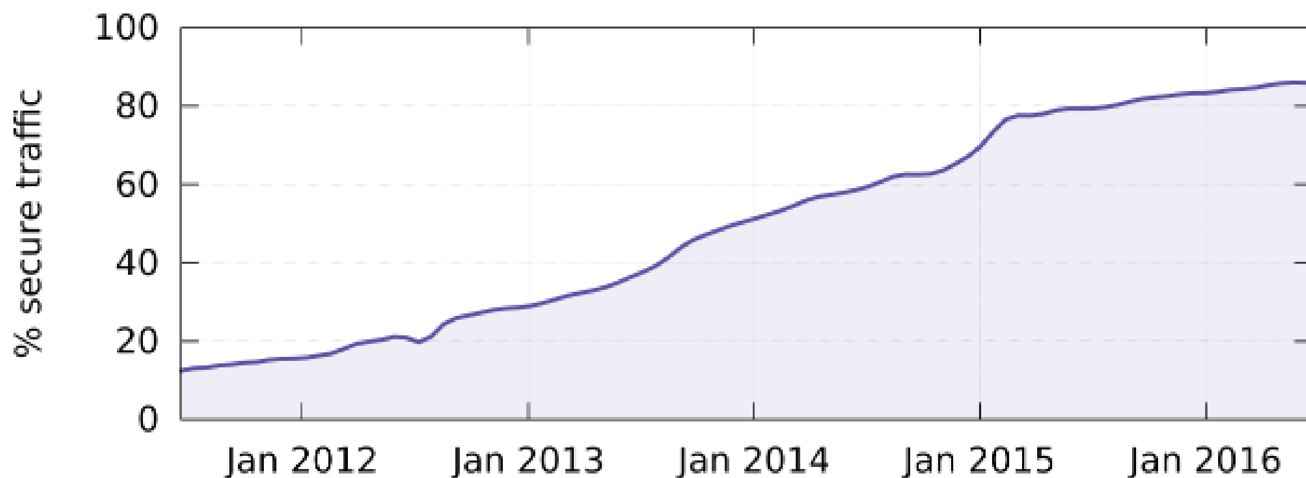


Figure 3: Increase in secure web traffic to Google's front-end servers.

1. Introduction

- 传统 HTTPS 栈的“握手延迟税”： RTT 被层层叠加
  - 传统栈中：TCP 建连通常至少 1 个 RTT 才能开始发应用数据
  - TLS 在其上继续叠加：TLS 还要再增加 2 个 RTT（经典 TLS）
  - 对 Web 来说，很多连接/事务是短传输，最受“额外握手 RTT”影响
  - 结论：想进一步压缩时延，**“减少握手 RTT”**是高优先级方向

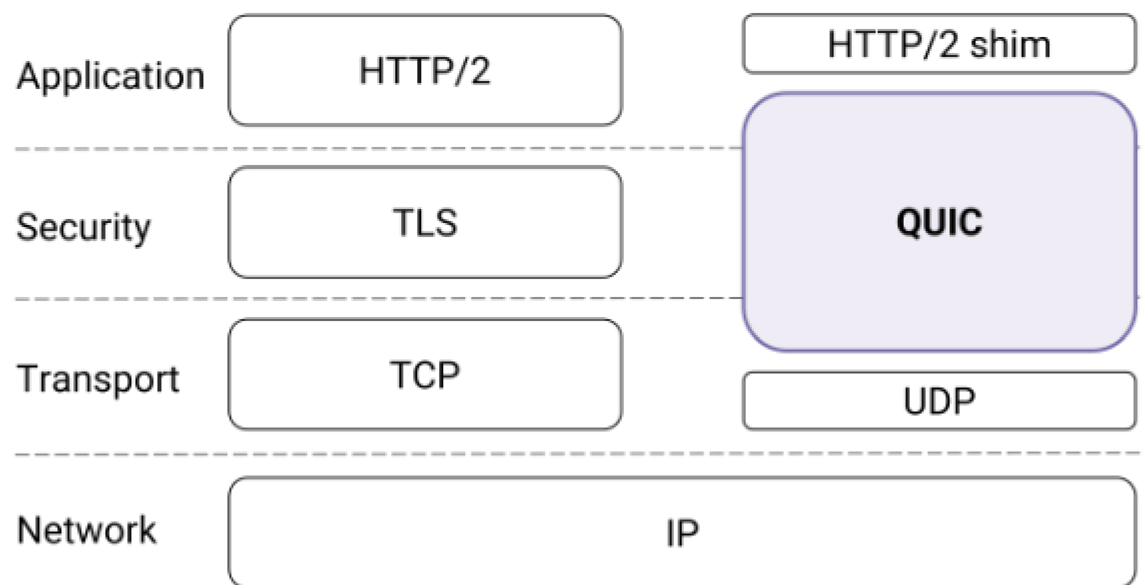


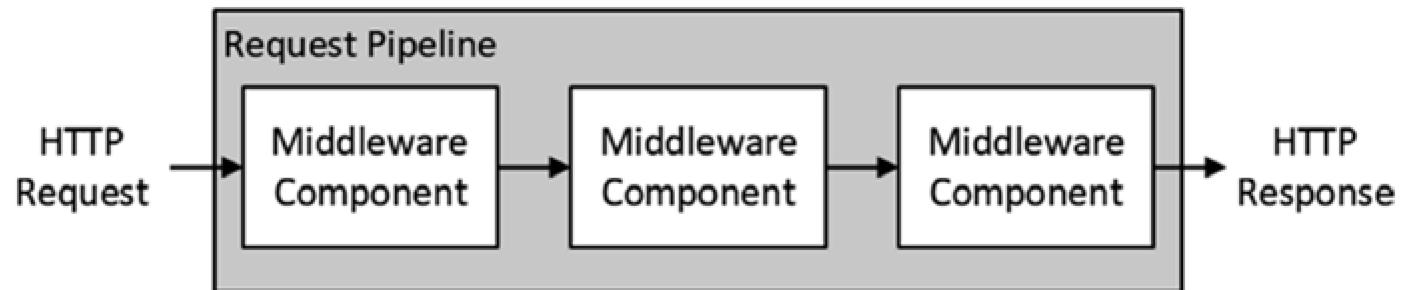
Figure 1: QUIC in the traditional HTTPS stack.

1. Introduction

- HTTP/2 + TCP 的结构性问题：Head-of-Line Blocking（队头阻塞）
  - 为减少多连接开销，HTTP/2 倾向单连接复用多个对象
  - 但 TCP 是字节流(bytestream)：应用难控制分帧，丢包导致后续数据即使到了也得等重传 → 形成“延迟税”
  - 这就是 TCP 层面的 **Head-of-line blocking** delay：一个丢包会拖慢同连接内其他对象/请求
  - QUIC 的方向：用 streams 做连接内多路复用，让“丢一个包”只阻塞相关 stream

1. Introduction

- TCP/TLS： 协议“固化/僵化”(ossification)
  - TLS（传输层安全协议）是一种用于在网络通信中提供加密、身份认证和数据完整性的安全协议，确保数据在传输过程中不被窃听或篡改。
  - Protocol Entrenchment（协议层固化）：新传输层协议很难大规模部署，很多网络设备默认“陌生即拦截”
  - 中间件（防火墙/NAT）成为事实上的控制点：NAT 会改写传输头部，防火墙会阻断未知流量
  - TCP 头部等未端到端保护的字段会被检查/修改，导致即使改 TCP 也很难，部署周期可能以十年计



1. Introduction

- 内核 TCP 的迭代周期太慢
  - Implementation Entrenchment（实现层固化）：TCP 通常在 OS 内核里，实现更新往往需要 OS 升级
  - OS 升级链路谨慎且影响系统全局：客户端可能落后多年，服务器升级也可能要数月
  - QUIC 的关键策略：user-space 实现，可以跟应用版本一起快速迭代、上线实验
  - 同时使用 UDP 承载来更容易穿越中间件
  - 并且 QUIC 强加密/认证，减少中间件篡改与进一步僵化的风险

1. Introduction

- QUIC 为什么“值得做”
  - QUIC 的“对症下药”：加密传输 + 更低握手时延 + streams 解决 HoL
  - **真实部署**规模（不是纸上协议）：
    - 在 Google 大规模部署，桌面搜索平均延迟降低 8.0%，移动端降低 3.6%
    - YouTube rebuffer rate 桌面降低 18.0%，移动降低 15.3%
    - 占 Google egress 超过 30%，估算约占全球互联网流量 7%
  - 标准化路径：先在生产迭代验证，随后进入 IETF 推进标准化

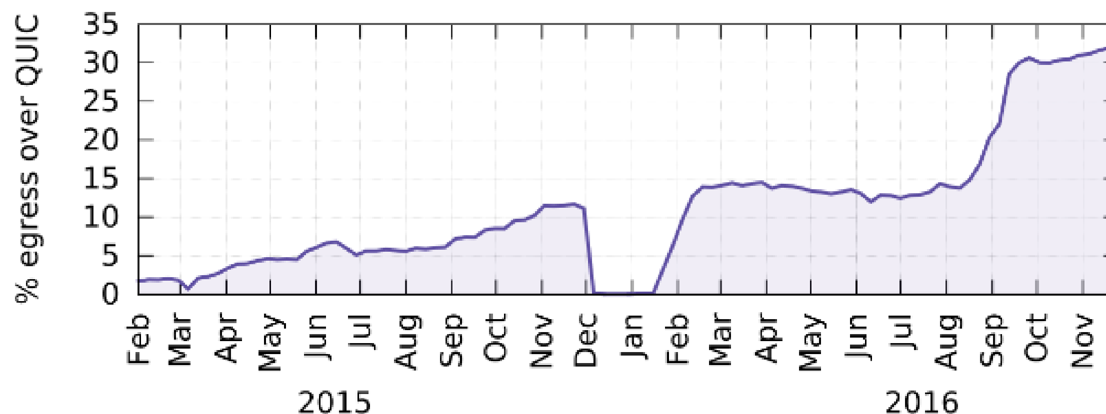


Figure 2: Timeline showing the percentage of Google traffic served over QUIC. Significant increases and decreases are described in Section 5.1.

Outline

I. Introduction

II. QUIC DESIGN AND IMPLEMENTATION

III. EXPERIMENTATION FRAMEWORK

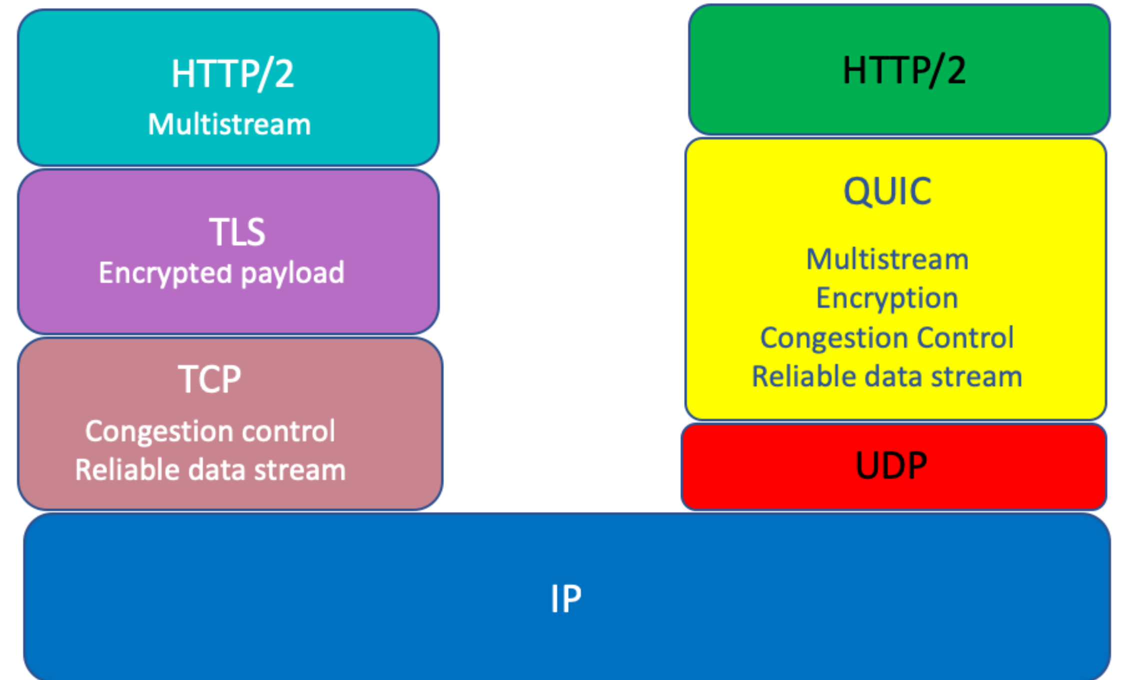
IV. INTERNET-SCALE DEPLOYMENT

V. QUIC PERFORMANCE

VI. EXPERIMENTS AND EXPERIENCES

3 QUIC DESIGN AND IMPLEMENTATION

- Overview
  - QUIC 为了“可部署 + 安全 + 低延迟”做了哪些关键设计：握手、流、加密、丢包恢复、迁移、流控/拥塞控、发现机制等
  - QUIC 把加密握手和传输握手合并、把多路复用放到传输层“流”里、并用加密减少中间盒篡改/僵化
  - 接下来聚焦：设计细节，以及为什么这些机制能解决 TCP/TLS 的关键痛点

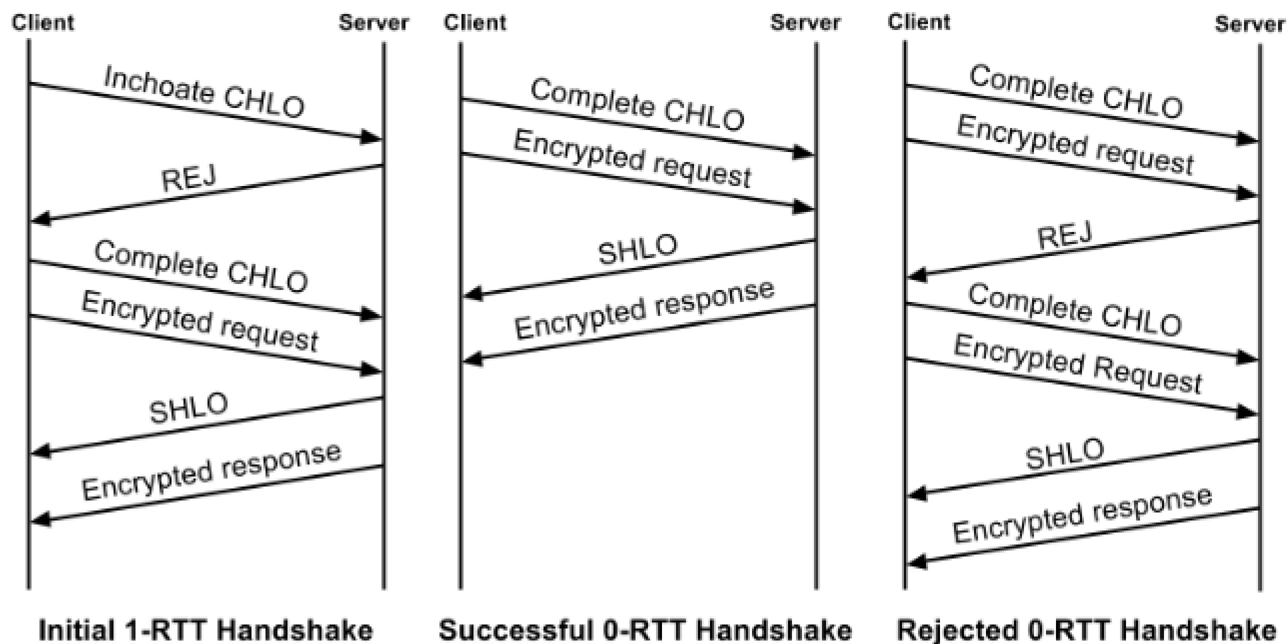


3 QUIC DESIGN AND IMPLEMENTATION

- Overview
  - 减少握手 RTT：合并握手，支持 0-RTT
  - 避免 HoL blocking：每个请求/响应用独立 stream，多路复用但互不阻塞
  - 抗中间件篡改/僵化：包加密/认证，尽量隐藏头部
  - 更好的丢包恢复：唯一包号 + ACK 显式 RTT 信息，避免 TCP 重传歧义
  - 连接迁移：用 Connection ID 识别连接而不是 5-tuple
  - 流控/拥塞控：流控限制接收缓冲；拥塞控可插拔便于实验

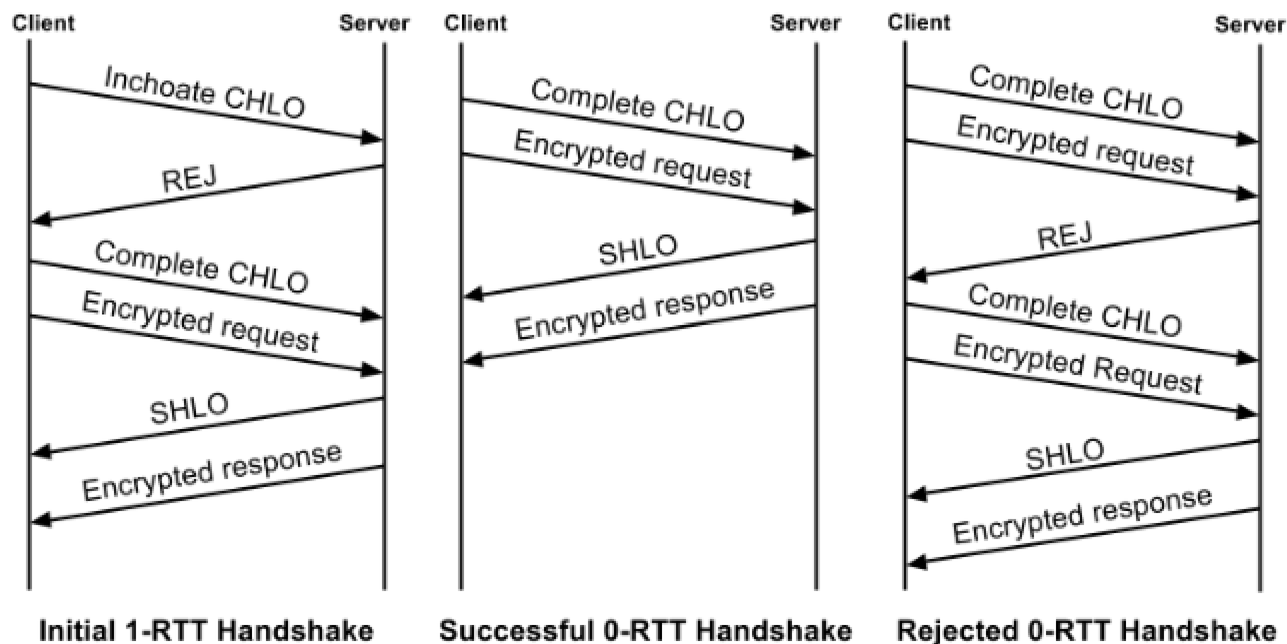
3 QUIC DESIGN AND IMPLEMENTATION

- 3.1 连接建立：1-RTT vs 0-RTT
  - 传统 HTTPS = TCP + TLS + HTTP/2，握手层层叠加，短连接开销大
  - QUIC 把“建立连接”和“建立密钥”合成一套流程，并且支持 0-RTT：如果客户端之前连过同一 origin，会缓存服务器信息，下次可以客户端发出握手包后立刻发应用数据（不用等服务器回包）

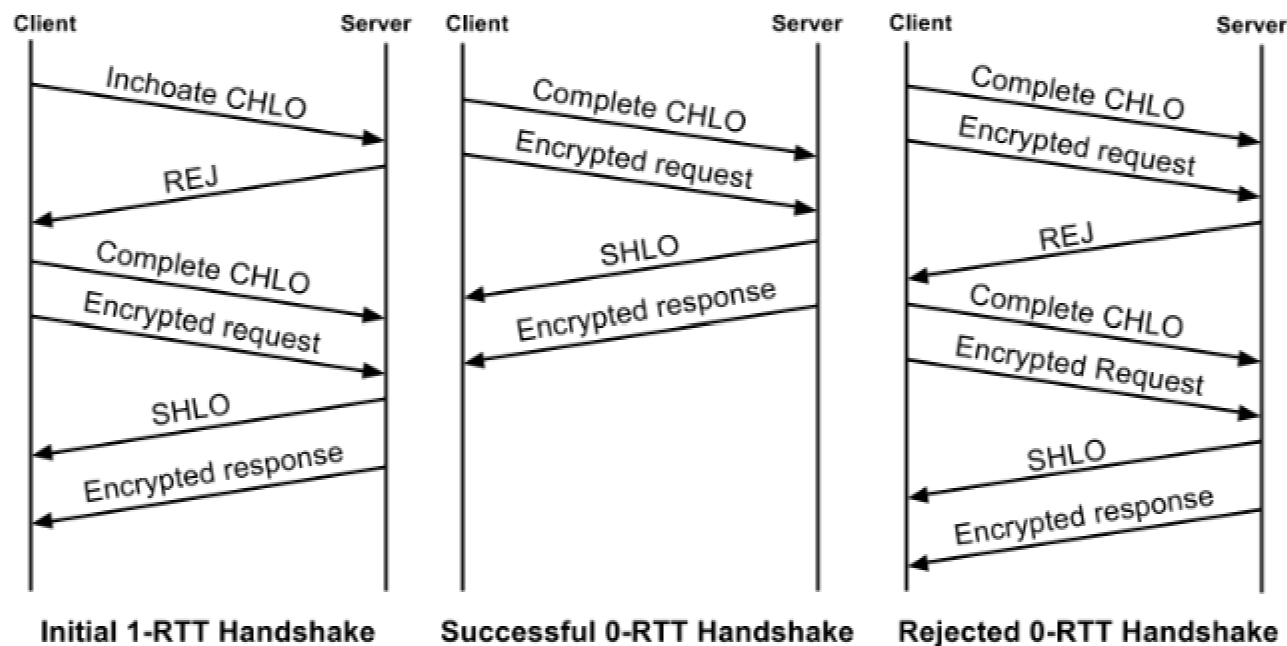


➢ 3.1 连接建立：1-RTT vs 0-RTT

- ➢ 客户端发 complete CHLO 后就能算出共享密钥，所以可以开始发“用初始密钥加密的数据”；服务器回 SHLO 后双方切到“前向安全密钥”



- 3.1 连接建立：1-RTT vs 0-RTT
  - 客户端第一包就“大胆猜一个版本”。如果服务器不支持，会回版本列表，导致多一次 RTT；设计上希望多数时候猜中，避免额外延迟，同时把版本信息喂进密钥派生，防降级攻击

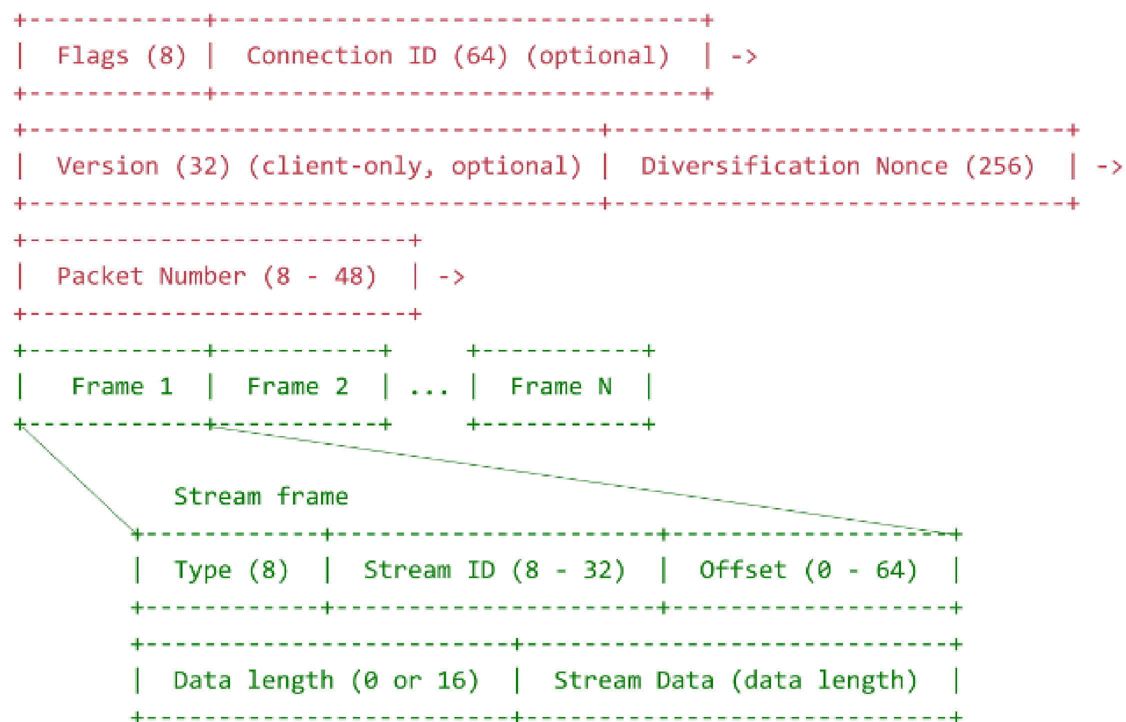


3 QUIC DESIGN AND IMPLEMENTATION

- 3.2 多路复用
  - Streams 是什么样的抽象
    - stream 是轻量的、可靠的双向字节流
    - stream id：客户端发起用**奇数**，服务器发起用**偶数**，避免冲突
    - 新建 stream 是“隐式的”（第一次在新 stream id 上发数据就算创建）
    - FIN 表示 stream 结束；也可以 cancel 某个 stream 而不断开整条连接（并且取消后不再重传该 stream 的数据）

3 QUIC DESIGN AND IMPLEMENTATION

- 3.2 多路复用
  - 一个 QUIC 包里可以装多个 stream 的 frame
    - 包 = 公共头 + 一个或多个 frames；一个包**能混装**多个 streams 的数据
    - 调度上，Google 的实现直接复用 HTTP/2 的优先级来决定先发哪个 stream

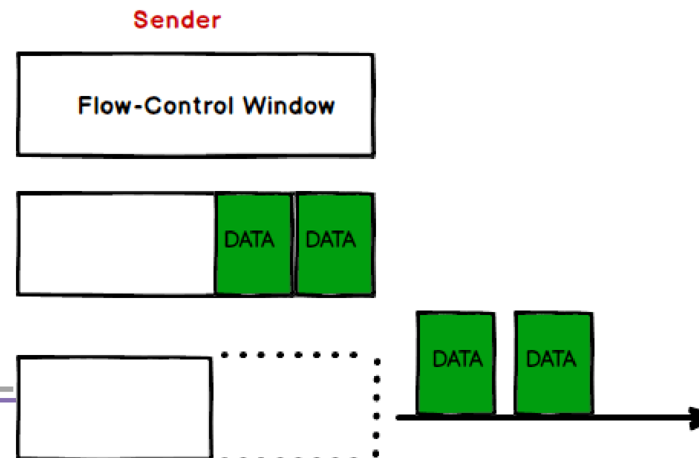


- 3.2 多路复用
  - 一个 QUIC 包里可以装多个 stream 的 frame
    - 为避免 TCP 顺序交付带来的 HoL：QUIC 在一个连接内支持多个 stream，丢一个 UDP 包只影响其中包含数据的 stream，其它 stream 的后续数据仍可交付
    - stream 是可靠双向字节流：stream ID：客户端奇数、服务端偶数
    - 创建是隐式的（首次发送即创建）；结束用 FIN；也支持取消 stream 而不拆连接

- 3.3 认证与加密
  - 除了少数早期握手包/重置包，QUIC 包都认证且大多加密
  - 头部只留必要字段明文：Flags、Connection ID、Version、nonce、Packet Number 等，用于路由/解密
  - 核心动机：防中间盒篡改与协议僵化
  - 同时，减少可见字段还能降低被动观察与指纹识别的空间，把网络里必须明文暴露的内容压到最少（仅保留路由与解密必需的少数公开字段）

- 3.4 丢包恢复
  - TCP 把“可靠性 + 交付顺序”绑在 sequence number 上，会出现重传歧义：ACK 到底确认的是原包还是重传包？导致一些丢包只能靠昂贵 timeout
  - QUIC 用“单调递增、每个包唯一”的 packet number，**即使重传也用新包号**，避免 TCP 的“重传歧义”
  - 交付顺序改用 stream offset 来表达，packet number 负责时间顺序 → loss detection 更简单更准
  - ACK 里还显式带了“收包到发 ACK 的延迟”，帮助更准的 RTT 估计

- 3.5 流量控制
  - 问题：某条 stream 的应用读得慢，会占满连接接收缓冲，导致其它 streams 发不进来（跨 stream 的 HoL）
  - 方案：同时做
    - connection-level flow control（限制**总**缓冲）
    - stream-level flow control（限制**单 stream** 缓冲）
  - 机制：信用式（credit-based），接收端用 window update 提升可接收 offset 上限



3 QUIC DESIGN AND IMPLEMENTATION

- 3.6 拥塞控制
  - 协议层不强依赖某个拥塞控制算法；实现提供可插拔接口，便于实验
  - 部署中 QUIC 和 TCP 都用 Cubic；但为匹配“视频客户端用两条 TCP 连接”的公平性，QUIC 在拥塞避免阶段使用 Cubic 的 mulTCP 变体
    - CUBIC 是一种拥塞控制算法，通过用“立方函数”来调节发送速率，使 TCP 和 QUIC 在高带宽、长时延网络中能更快、更稳定地利用可用带宽
  - 视频客户端通常会用两条并行 TCP 连接来下载视频（吞吐更稳、更抗波动），如果 QUIC 只用一条连接，可能在“与两条 TCP 竞争带宽”时吃亏。于是 QUIC 的拥塞避免阶段用了 CUBIC 的 mulTCP 变体，让“一条 QUIC 连接（两个 streams）”在带宽竞争中更接近“两条 TCP”的行为

- 3.7 NAT Rebinding and Connection Migration
- 问题：UDP 更容易遇到 NAT rebinding
  - NAT 通常会给一个“UDP 映射”较短超时；当一段时间不发包、或者 NAT 端口重分配时，同一个客户端可能突然变成新的外部端口。对传统 TCP 来说，连接识别依赖 5-tuple（源/目的 IP+端口+协议），一变就相当于断了
- QUIC 的关键设计：Connection ID
  - QUIC 用一个 64-bit Connection ID 来识别连接，而不是只靠 5-tuple。这样当客户端 IP/端口变化（NAT rebinding、切网）时，只要后续包还带着同一个 Connection ID，服务端就能把它**归到原连接**上

3 QUIC DESIGN AND IMPLEMENTATION

- 3.7 NAT Rebinding and Connection Migration
- “NAT rebinding” vs “连接迁移”
  - NAT rebinding：通常是“网络在背后悄悄换了端口/映射”，客户端未必主动。论文说 QUIC 的设计可以“解决 NAT rebinding”
  - 连接迁移（mobility）：更强的场景：客户端主动从 Wi-Fi 切 4G/5G，IP 真的变了。论文提到当时“客户端迁移（client mobility）仍在推进中”
- “连接能换地址继续用”也意味着：需要机制避免被攻击者拿着 Connection ID 冒充/劫持
  - 所以 QUIC 里会有路径验证等思路

3 QUIC DESIGN AND IMPLEMENTATION

- 3.8 QUIC Discovery for HTTPS
- 客户端第一次访问，怎么知道服务器支不支持 QUIC?
  - 答案：不知道。所以第一次还是走 TCP+TLS。然后服务器在 HTTPS 响应里用 Alt-Svc 头告诉客户端：“这个 origin 也支持 QUIC，以后可以试”
- 后续访问：QUIC 与 TCP “赛跑”（racing）
  - 客户端之后会倾向 QUIC，同时尝试 QUIC 与 TLS/TCP，但 TLS/TCP 会被最多延迟 300ms，让 QUIC 优先
  - 如果 QUIC 被阻断（某些网络/防火墙不让 UDP）或者握手包超过路径 MTU 导致握手失败，就回退到 TLS/TCP

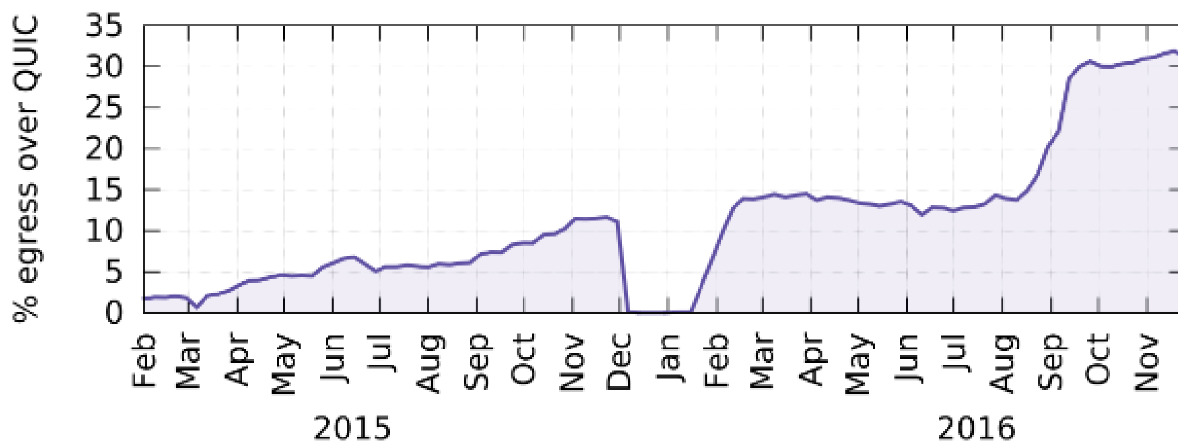
- 3.8 QUIC Discovery for HTTPS
- 为什么这个发现机制很关键
  - 它把“部署风险”压到最小：网络环境复杂，UDP 不一定通；必须有可靠回退
  - 它让 QUIC 能逐步提升覆盖率：越多成功样本→越能推动优化与运维策略（比如后面的UDP proxying）

Outline

I. Introduction
II. QUIC DESIGN AND IMPLEMENTATION
III. EXPERIMENTATION FRAMEWORK
IV. INTERNET-SCALE DEPLOYMENT
V. QUIC PERFORMANCE
VI. EXPERIMENTS AND EXPERIENCES

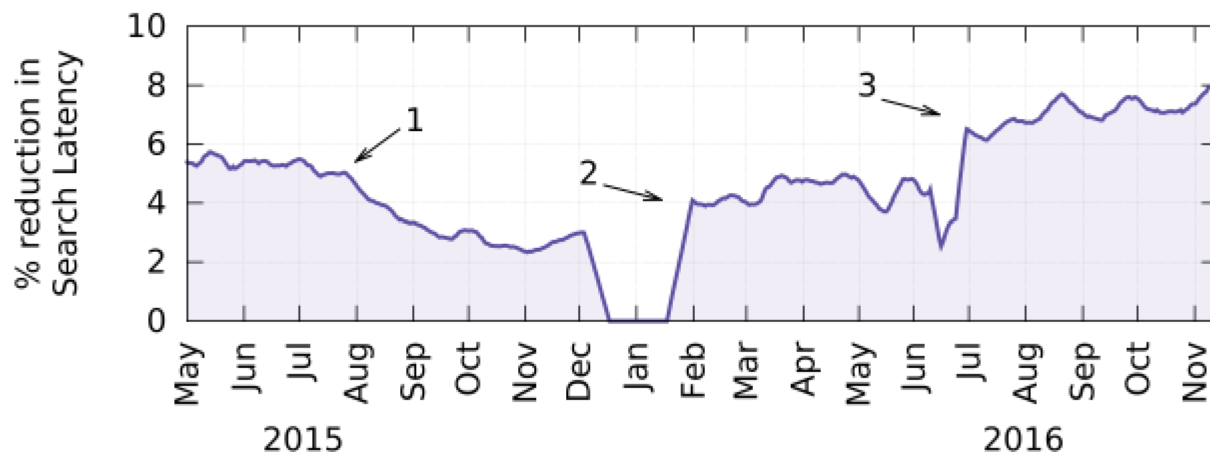
4 EXPERIMENTATION FRAMEWORK

- "互联网规模"的持续实验
  - QUIC 的开发高度依赖持续的互联网规模实验：用真实用户/真实网络来验证"某个特性是否真的有价值"、以及"参数该怎么调"
  - 目标不只是"协议层指标更好看"，而是要看最终用户/应用指标：比如搜索响应时间、视频卡顿率
  - 这种方式带来一个关键好处：QUIC 能通过小步快跑（small but repeatable improvements）累积出长期、稳定的性能提升轨迹



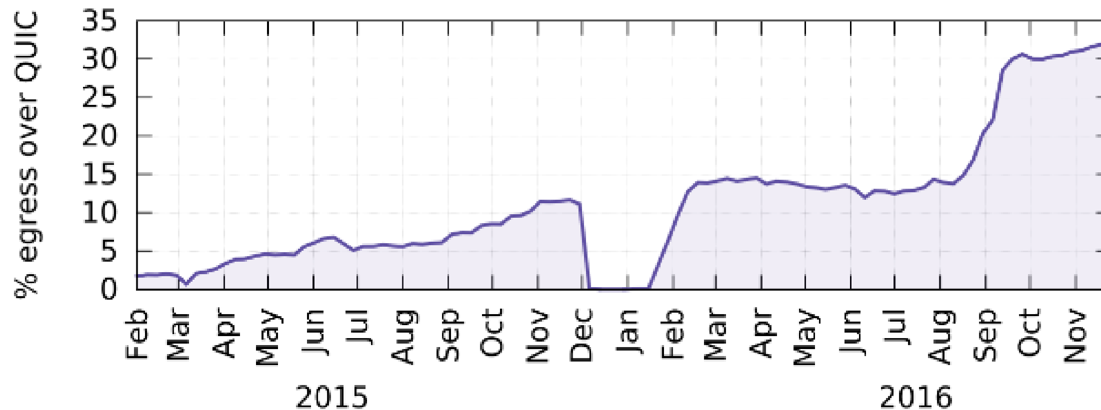
4 EXPERIMENTATION FRAMEWORK

- Chrome 侧实验框架
  - Chrome 有强实验/分析框架：把用户伪随机分配到不同实验组，允许在全量前先做 A/B test
  - 指标覆盖面很广：从 HTTP 错误率到传输握手延迟。既看“能不能用”，也看“快不快”
  - 参与统计上报的客户端会回传：指标 + 自己属于哪些实验组，分析时能按实验维度切分
  - 安全阀：可以快速禁用某个实验，保护用户免受“坏实验”影响



4 EXPERIMENTATION FRAMEWORK

- Server 侧实验框架：全局放量要靠 Feature Toggle + 监控告警
  - Google 前端服务器是全球分布的大规模集群：数据中心 + ISP 网络内部都有
  - 它们同时终止 TLS/TCP 和 QUIC 连接、并做负载均衡
- 服务端怎么做到“安全地做协议实验”
  - 能在每台服务器上 toggle（开/关）特性：这意味着出问题可以快速阻止
  - 这个机制使得全球实验受控，同时把大规模宕机风险压到很低
  - 服务端还会对当前/历史 QUIC 连接上报性能数据，集中监控系统做聚合、可视化、告警



Outline

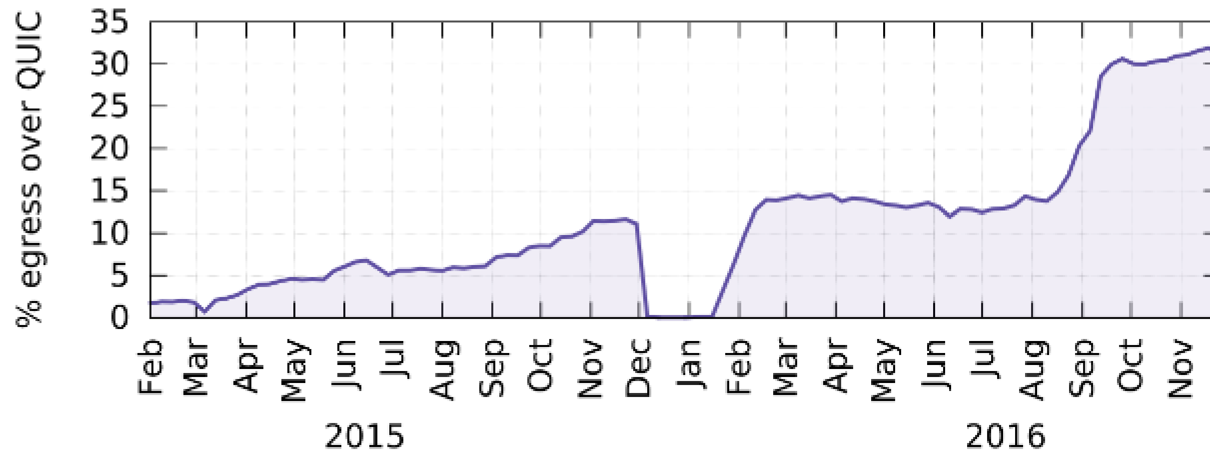
I. Introduction
II. QUIC DESIGN AND IMPLEMENTATION
III. EXPERIMENTATION FRAMEWORK
IV. INTERNET-SCALE DEPLOYMENT
V. QUIC PERFORMANCE
VI. EXPERIMENTS AND EXPERIENCES

5 INTERNET-SCALE DEPLOYMENT

- QUIC 如何做到“互联网规模部署”
  - 第 4 部分的实验框架，使 QUIC 可以安全地做全球部署
    - QUIC 的部署不是一条直线，而是一条“放量—回滚—修复—再放量”的曲线
    - 2013 年 6 月：Chrome 加入 QUIC 支持，但一开始只是开发者用命令行开关打开
    - 2014 年初：开始用 Chrome 的实验框架做线上试验，最初放量非常小（<0.025%），之后逐步扩大
    - 2017 年 1 月：Chrome 与 Android YouTube app 上几乎所有用户默认启用 QUIC
  - 关键问题：在真实互联网环境里，QUIC 的占比是怎么逐步上去的？为什么中间会“突然掉到 0”？

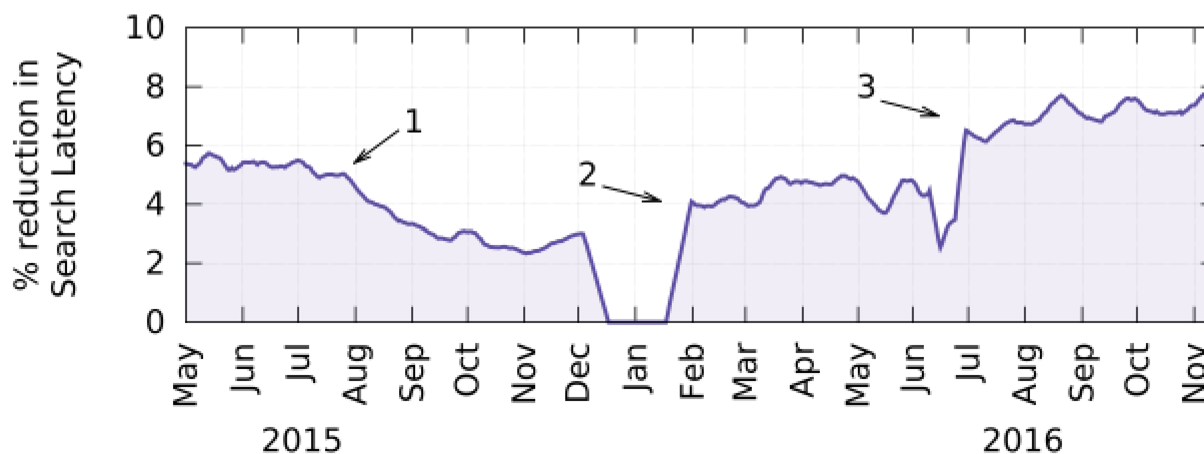
5 INTERNET-SCALE DEPLOYMENT

- 为什么 Figure 2 会出现“突然掉到 0”
  - 这才是工程里最关键：出现安全风险时，他们会果断全局关掉 QUIC
  - 2015 年 12 月：发现一个实现漏洞：极少数情况下 0-RTT 请求可能未加密
  - 处理：服务器端全球禁用 QUIC（所以 Figure 2 上会看到流量占比掉到接近 0），修复后随着客户端更新再恢复
  - 2016 年 9 月：YouTube app 开始使用 QUIC，Google egress 里的 QUIC 占比从 ~15% 上升到 >30%
    - 移动端视频流量太大，一旦启用就会“改写曲线”



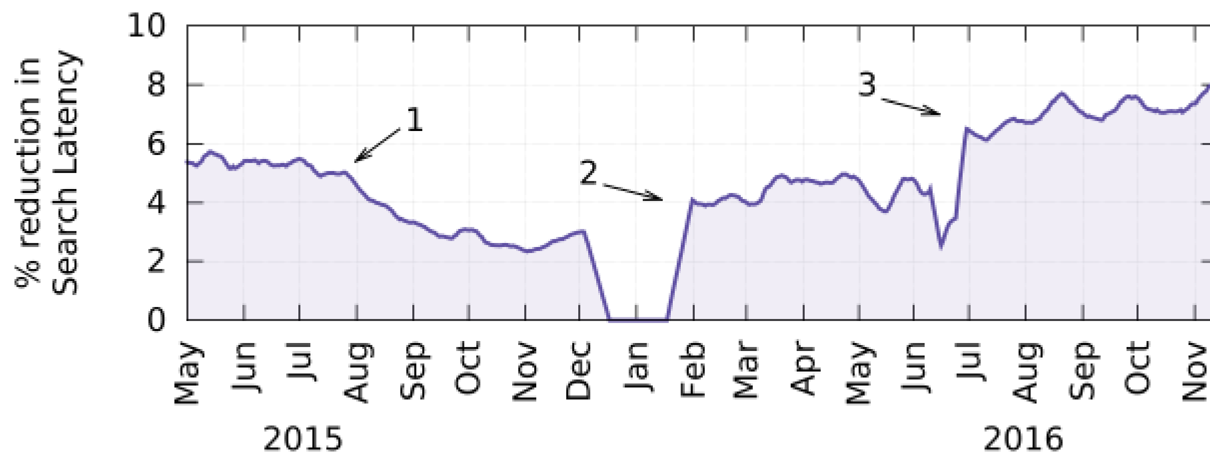
5 INTERNET-SCALE DEPLOYMENT

- 典型监控指标：Search Latency
  - Search Latency：从用户输入搜索词开始，到所有搜索结果内容生成并送达为止的时间（包括图片/嵌入内容）
    - 这是端到端指标，覆盖了网络、服务端、渲染等多因素，但也正因为如此，它反映“用户体验”
  - 论文用 Figure 6 跟踪了 18 个月 QUIC 相对 TCP/TLS 的搜索延迟改善趋势，并明确指出期间有两次回退和一次改善



5 INTERNET-SCALE DEPLOYMENT

- 回退 1（2015/07 起约 5 个月）
  - 原因：基础设施变化 + 客户端配置 bug
- 回退 2（2015/12）：对应我们刚才说的安全漏洞→全球禁用 QUIC
- 改善点（2016/07）：UDP proxying 让收益从 ~4% 提升到 >7%
  - 背景：一些 ISP 内的边缘点（REL）因为限制不能终止 TLS，只能做 TCP 代理；而 QUIC “加密与传输绑在一起”，不能用 TCP 那套方式，于是他们部署了 UDP proxying：把 UDP 包转发到能终止 QUIC 的前端服务器
  - 结果：Search Latency 的平均改善从约 4% 提升到 7% 以上



Outline

I. Introduction

II. QUIC DESIGN AND IMPLEMENTATION

III. EXPERIMENTATION FRAMEWORK

IV. INTERNET-SCALE DEPLOYMENT

V. QUIC PERFORMANCE

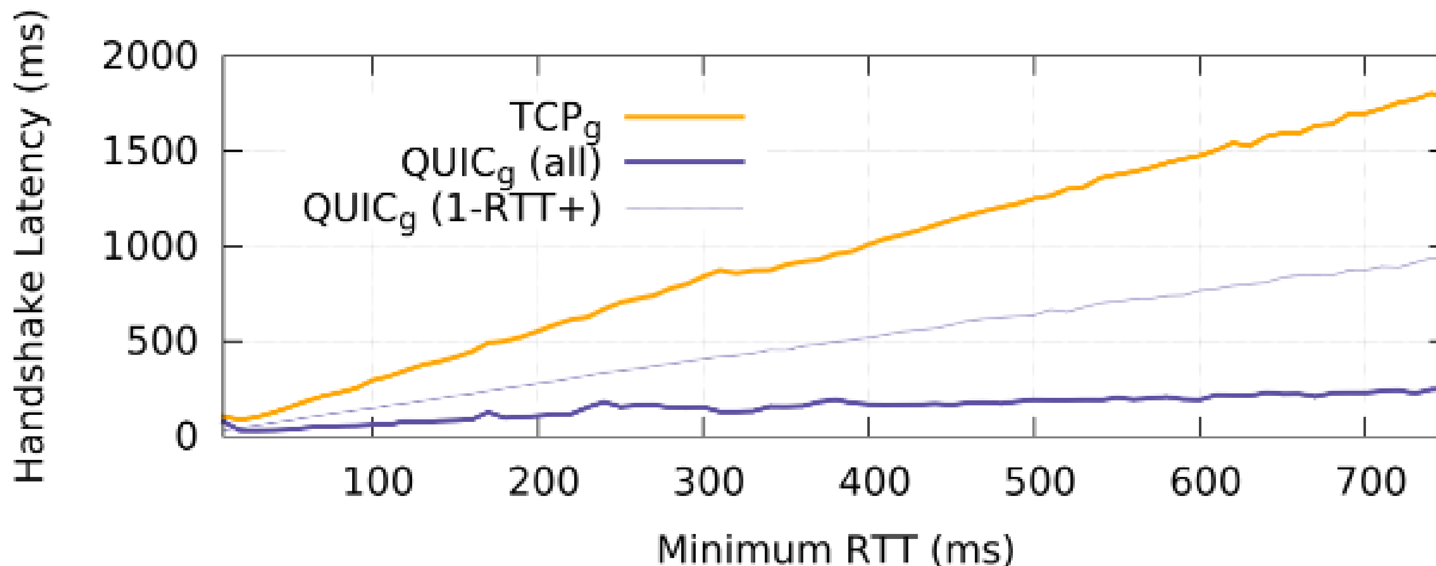
VI. EXPERIMENTS AND EXPERIENCES

6 QUIC PERFORMANCE

- 对照组/实验组、样本量与时间窗
  - 实验把用户随机分成：
    - QUICg（实验组，允许用 QUIC；也包含少量因握手失败等回落到 TCP 的请求）
    - TCPg（对照组，仅 TLS/TCP）
    - 协议/实现条件尽量“对齐”：两边都用 paced Cubic；搜索侧 HTTP/2 over single TLS/TCP；视频侧在非 QUIC 客户端通常用两条 TCP 连接拉取音视频数据等
  - 样本规模与采集窗口：
    - 使用 QUIC v35，样本“over a billion”级别
    - 搜索数据：2016-12-12 ~ 2016-12-19；视频数据：2017-01-19 ~ 2017-01-26

6 QUIC PERFORMANCE

- 关键机制收益——握手延迟几乎“抹平 RTT”
  - QUIC 把加密与传输握手结合；如果是 0-RTT，作者在服务端统计口径里把握手延迟记为 0ms
- Figure 7给出最直观对比：
  - TLS/TCP：RTT 越大，握手延迟几乎线性上升
  - QUIC：曲线“几乎不随 RTT 增长”，因为大量连接走 0-RTT



6 QUIC PERFORMANCE

- 搜索延迟收益
  - Search Latency定义：从用户输入搜索词到搜索结果内容生成并送达（含图片/嵌入内容）的端到端时间
  - Table 1（搜索部分）核心结论：Desktop：平均 降低 8.0%；99 分位可到 降低 16.7%Mobile：平均 降低 3.6%；99 分位可到 降低 14.3%
- RTT 越高，省下握手 RTT 的价值越大；并且高 RTT 往往伴随更差网络质量，QUIC 的改进丢包恢复也可能贡献更明显

	% latency reduction by percentile							
		Lower latency						Higher latency
	Mean	1%	5%	10%	50%	90%	95%	99%
Search								
Desktop	8.0	0.4	1.3	1.4	1.5	5.8	10.3	16.7
Mobile	3.6	-0.6	-0.3	0.3	0.5	4.5	8.8	14.3
Video								
Desktop	8.0	1.2	3.1	3.3	4.6	8.4	9.0	10.6
Mobile	5.3	0.0	0.6	0.5	1.2	4.4	5.8	7.5

6 QUIC PERFORMANCE

- 视频体验——延迟下降 + 卡顿率显著下降
  - Video Playback Latency（点播放到开始播放）：Table 1 显示 QUIC 对视频启动也有明显改善：Desktop：平均 降低 8.0%Mobile：平均 降低 5.3%
  - Video Rebuffer Rate（卡顿/缓冲占比）：Table 2 显示 QUIC 对卡顿改善更“硬核”：Desktop：平均 降低 18.0%，在更高分位也有明显下降 Mobile：平均 降低 15.3%（并在较高分位仍有改善）
- 机制解释：
  - 卡顿率对“握手”不敏感，更受丢包恢复延迟与吞吐/窗口限制影响
  - 视频在 TCP 下常用两条连接，导致丢包恢复更慢、尾部恢复概率更高；QUIC 在单连接多流里更容易降低这种尾部恢复代价

		% rebuffer rate reduction by percentile				
		Fewer rebuffers				More rebuffers
	Mean	< 93%	93%	94 %	95%	99%
Desktop	18.0	*	100.0	70.4	60.0	18.5
Mobile	15.3	*	*	100.0	52.7	8.7

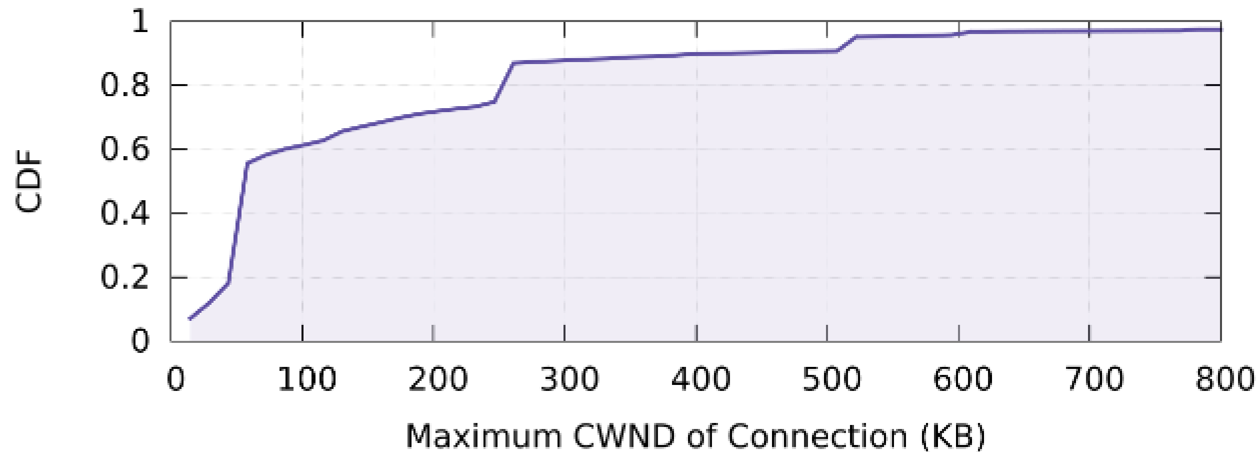
6 QUIC PERFORMANCE

- 不同地区收益不同，原因：
  - South Korea：RTT 低、丢包低 → QUIC 改善更接近 TCP（收益较小）
  - USA：中等网络条件 → 改善更明显
  - India：RTT 高、重传高 → 收益最大
- QUIC 的优势在“高 RTT + 高丢包/拥塞”的真实互联网环境里更容易兑现

			% Reduction in Search Latency		% Reduction in Rebuffer Rate	
Country	Mean Min RTT (ms)	Mean TCP Rtx %	Desktop	Mobile	Desktop	Mobile
South Korea	38	1	1.3	1.1	0.0	10.1
USA	50	2	3.4	2.0	4.1	12.9
India	188	8	13.2	5.5	22.1	20.2

6 QUIC PERFORMANCE

- 代价与边界
  - TCP 接收窗口限制：作者发现视频 TCP 连接里有一部分会被客户端接收窗口卡住：他们检查 2016 年 3 月的一周视频连接，约 4.6% 连接是 receive-window-limited；很多被限制在 64KB 量级的最大接收窗口（还讨论了可能与 window scaling 缺失/中间件干扰有关）
  - 对照之下，QUIC 客户端默认连接级流控上限 15MB，更不容易在这类场景下成为瓶颈

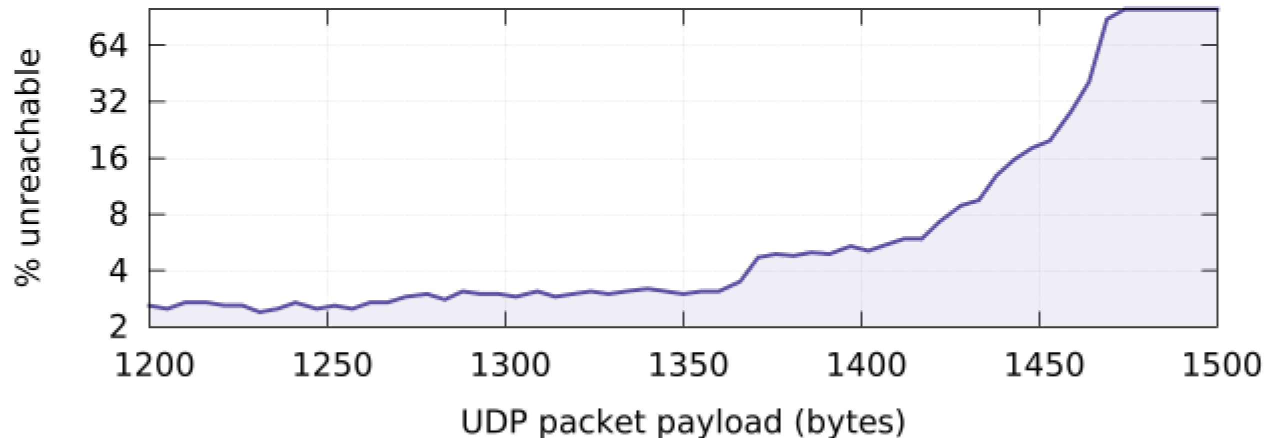


Outline

I. Introduction
II. QUIC DESIGN AND IMPLEMENTATION
III. EXPERIMENTATION FRAMEWORK
IV. INTERNET-SCALE DEPLOYMENT
V. QUIC PERFORMANCE
VI. EXPERIMENTS AND EXPERIENCES

7 EXPERIMENTS AND EXPERIENCES

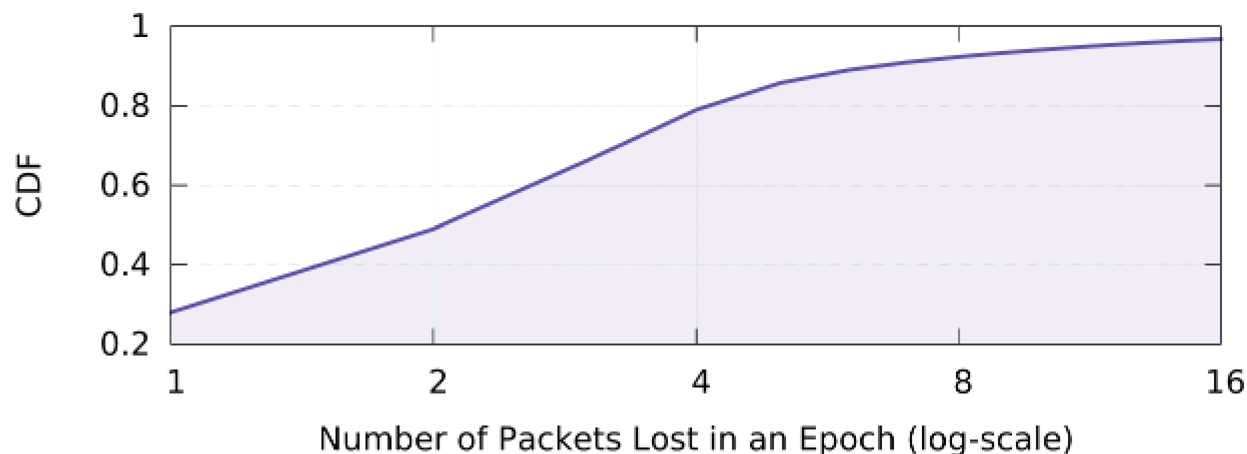
- 最大包大小（Path MTU 的现实约束）
  - 让 Chrome 客户端对 echo server 发送不同 UDP payload（1200~1500 bytes），统计“收不到回包就算失败”
  - 结果显示 1450 bytes 以后不可达迅速上升（加上 UDP/IP 头后超过 1500 以太网 MTU）
  - 因此选择 1350 bytes 作为 QUIC 默认 payload，并提到未来会考虑 MTU discovery



- UDP Blockage / Throttling
  - 以 2016.11 的视频回放数据衡量：95.3% 尝试 QUIC 的客户端能成功使用 QUIC
  - 4.4% 无法使用 QUIC（QUIC/UDP 被 block，或路径 MTU 太小）；常见于企业网防火墙环境；没看到整个 ISP 全面封 UDP/QUIC
  - 还有 0.3% 看起来对 QUIC/UDP 限速：在高峰期出现更高丢包和更低带宽；Google 会对整个 AS 临时禁用 QUIC 并联系运营方，限速 AS 的比例从 2015.6 的 1% 降到 2016.11 的 0.3%

7 EXPERIMENTS AND EXPERIENCES

- FEC 为什么不值得
  - 核心问题：丢包恢复能不能靠 FEC 提升尾延迟？作者做了实验，结论是：收益不够稳定 + 代价不小，最后移除
  - FEC 用冗余让接收端在不重传的情况下恢复丢包；他们实验了 XOR parity（简单校验），目标是恢复“单包丢失”
  - 实验结果：重传少了，但用户体验指标没变好甚至更差



作者把每次“持续大约 1 个 RTT 的丢包段”当作一个 loss epoch，然后统计每个 epoch 里丢了多少个包，并画 CDF；结论是：**如果 FEC 只能恢复“1 个包丢失”，那它最多只能对不到 30% 的丢包事件起作用**（因为很多 epoch 里不止丢 1 个包）

总结

- 动机：Web 强依赖低延迟 + HTTPS 普及带来握手成本 + TCP/中间盒导致 HoL 与协议僵化 → 需要一个“能快、能改、能部署”的新传输层（QUIC）
- 机制：合并传输与加密握手并支持 0-RTT（减少 RTT 税）；用 streams 做多路复用（缓解 HoL）；包号唯一化 + ACK 设计改进丢包恢复；连接 ID 支持 NAT rebinding/迁移；头部尽量加密以避免中间盒“绑死协议”
- 工程：QUIC 不是“写完标准再部署”，而是靠 Chrome/服务端的 A/B 实验、监控与一键回滚，做到互联网规模迭代
- **QUIC 的最大价值不仅是更快的协议，而是一个“可持续演进、可观测、可回滚”的传输平台**