smoltcp BBR 拥塞控制实现

从算法到测试的完整实践

Jidong Chen

November 10, 2025

项目背景

什么是 BBR?

- Google 开发的现代拥塞控制算法
- 基于**速率**而非**丢包**
- 估计瓶颈带宽和 RTT，主动控制发送速率

为什么在 smoltcp 中实现 BBR?

- smoltcp：面向裸机/嵌入式的独立 TCP/IP 协议栈
- 原支持：Reno、Cubic（传统基于丢包的算法）
- 目标：在高带宽、高延迟网络中获得更好性能

项目成果

技术实现

- BBR 状态机（4 个状态）
- 带宽估计和 RTT 跟踪
- TCP Pacing 机制

测试框架

- 网络仿真（tc netem + tbf）
- 性能测试工具
- 可重复的测试流程

当前状态

性能与 Cubic 相当，BBR 预期优势尚未完全体现

关键收获

最重要的成果：

建立了完整的网络性能测试方法

- 使用 Linux tc 工具模拟真实网络环境
- 带宽延迟积（BDP）的计算和配置
- 设计测试场景对比不同拥塞控制算法
- 理解延迟、带宽、缓冲区的相互关系

价值

这套方法论对理解和调试拥塞控制算法至关重要，价值远超单个算法的实现。

传统算法 vs BBR

传统算法（Reno/Cubic）

- 基于丢包反馈
- 填满缓冲区直到丢包
- 然后退避（锯齿状）

问题：

- 高延迟（bufferbloat）
- 吞吐量不稳定

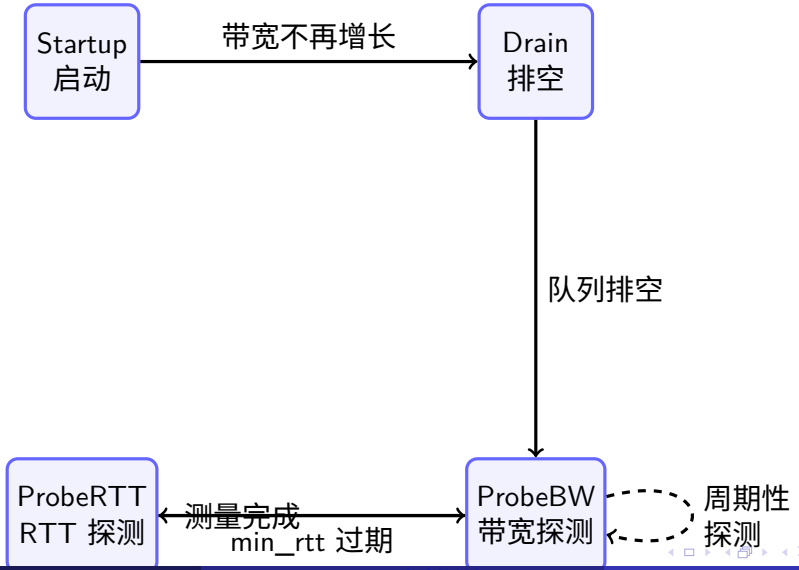
BBR

- 基于速率控制
- 估计瓶颈带宽
- 估计往返时延
- 主动避免队列堆积

优势：

- 低延迟
- 稳定吞吐量

BBR 状态机


Startup
启动
带宽不再增长
Drain
排空
队列排空
ProbeRTT
RTT 探测
测量完成
min_rtt 过期
ProbeBW
带宽探测
周期性
探测

BBR 状态详解

Startup（启动）

指数增长发送速率，快速找到可用带宽

Pacing gain $= 2.77$

Drain（排空）

排空启动阶段积累的队列

Pacing gain $= 0.75$

ProbeBW（带宽探测）

稳态运行，周期性探测更多带宽

8 轮循环：$1.25, 0.75, 1.0 \times 6$

ProbeRTT（RTT 探测）

降低拥塞窗口至最小（4 MSS）

测量真实传播延迟

BBR 核心机制

带宽估计

- 滑动窗口最大值滤波器（10 个 RTT）
- 计算：已确认字节数 / 已耗时间

RTT 跟踪

- 维护 10 秒窗口内的最小 RTT
- 过滤队列延迟，保留传播延迟

速率计算

$$\text{pacing\_rate} = \text{bandwidth} \times \text{pacing\_gain} \times (1 - \text{margin})$$

实现的旅程：意外的发现

1. **最初目标**：只实现 BBR 算法
   - 带宽估计
   - RTT 跟踪
   - 状态机
2. **研究发现**：通过阅读 Linux 内核代码和理解框架
3. **问题根源**：理解框架后确认
   - BBR 依赖 TCP Pacing 才能正常工作
   - smoltcp 原本没有 Pacing 机制
4. **解决方案**：先实现 TCP Pacing

为什么需要 Pacing?

问题：没有 Pacing 时

- 数据包在拥塞窗口允许时突发发送
- 突发流量瞬间填满瓶颈缓冲区
- 造成丢包和延迟增加
- **BBR 的速率控制形同虚设**

Pacing 的作用

- 根据计算出的速率均匀发送数据包
- 避免突发造成的缓冲区溢出
- 让 BBR 的速率控制真正生效

关键认识

Pacing 不是可选项，而是 BBR 的**必要前提**

Pacing 实现原理

基本机制

每个 socket 维护一个定时器 pacing_next_send_at

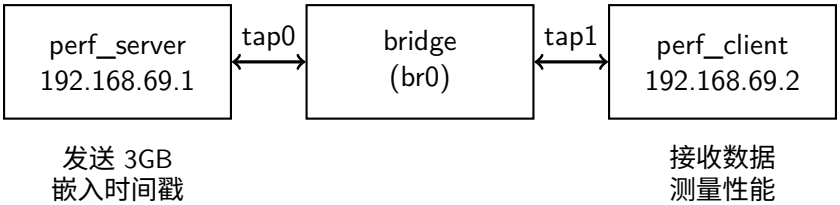
```
// After sending a packet
let pacing_rate = self.congestion_controller.pacing_rate();
if pacing_rate > 0 {
    let delay_micros = (packet_size * 1_000_000) / pacing_rate;
    let delay = Duration::from_micros(delay_micros);
    self.pacing_next_send_at = Some(now + delay);
}

// Before transmission
if let Some(next_send_at) = self.pacing_next_send_at {
    if now < next_send_at {
        return Ok(());  // Defer transmission
    }
}
```

Pacing 对不同算法的影响

算法	Pacing 状态	影响
BBR	启用	必需，否则无法正常工作
Cubic	禁用	零性能影响
Reno	禁用	零性能影响

- BBR：`pacing_rate()` 返回计算出的速率
- Cubic/Reno：`pacing_rate()` 返回 0（禁用）
- 传统算法依靠 ACK 时钟自然调节发送速率

网络拓扑



流量控制

每个 TAP 接口应用两层 qdisc：

- **netem**：添加延迟（模拟距离）
- **tbf**：限制带宽和缓冲区（模拟瓶颈）

流量控制命令详解

netem（网络模拟器）

```
tc qdisc add dev tap0 root handle 1: netem delay 10ms limit 4000
```

- 添加 10ms 传播延迟（RTT = 20ms）
- limit=4000：队列限制，避免提前丢包

tbf（令牌桶过滤器）

```
tc qdisc add dev tap0 parent 1:1 handle 10: tbf rate 400mbit
    burst 15000 limit 6000000
```

- rate=400Mbit：带宽限制
- burst=15000：允许短暂突发
- limit=6MB：**路由器队列**（测试关键!）

为什么这样配置缓冲区?

计算 BDP（Bandwidth-Delay Product）

$$\text{BDP} = \text{Bandwidth} \times \text{RTT} = 400\ \text{Mbit/s} \times 0.02\text{s} = 1\ \text{MB}$$

$$\text{Buffer} = 4000\ \text{packets} \times 1500\ \text{bytes} = 6\ \text{MB} = \mathbf{6} \times \text{BDP}$$

设计理由

1. **测试对比**：
   - Cubic/Reno：填满缓冲区，造成 bufferbloat
   - BBR：应该保持低队列，即使缓冲区很大

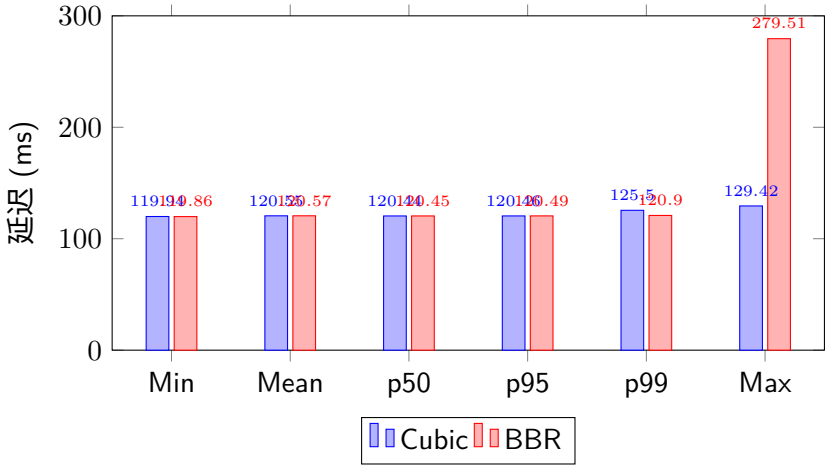
性能对比

指标	Cubic	BBR
吞吐量 (Gbps)	0.385	0.384
传输时间 (s)	66.96	67.14
带宽利用率	96%	96%
平均延迟 (ms)	120.552	120.571
p50 延迟 (ms)	120.435	120.449
p95 延迟 (ms)	120.458	120.488
p99 延迟 (ms)	125.502	120.895
最大延迟 (ms)	129.421	279.506

关键发现

- 吞吐量几乎相同
- 但都显示约 100ms 队列延迟（严重 bufferbloat）

延迟分布对比


300
200
100
0
延迟 (ms)
119
120
120
120
125
129.42
279.51
Min
Mean
p50
p95
p99
Max
Cubic
BBR

结果分析

1. 吞吐量相当

两种算法都能充分利用可用带宽（ 96%）

2. 缓冲区膨胀严重

- 基准 RTT：20 ms
- 实际延迟： 120 ms
- 队列延迟： 100 ms（远超预期）

3. BBR 未达预期

理论上 BBR 应保持低队列占用（ 2 BDP），但实际测试显示队列延迟与 Cubic 相当。

可能的问题：

- 带宽估计不够准确
- ProbeRTT 未正确清空队列

项目成果总结

实现成果

- 完整的 BBR 状态机
- 带宽估计和 RTT 跟踪
- TCP Pacing 框架

测试方法

- 网络仿真框架
- 完整测试流程

当前状态

BBR 实现完成，性能与 Cubic 相当，
但预期的延迟和队列管理优势尚未完全体现。

经验教训

1. 实现的复杂性

- BBR 不仅是算法，而是需要多个协同机制的系统
- Pacing 是必要前提，而非可选项

2. 测试环境的重要性

- 真实网络太不可控
- 仿真环境必须精确配置
- BDP 是关键参数，必须正确计算

最宝贵的收获

建立了完整的测试和调试方法论

1. **如何搭建网络仿真环境**
   - TAP 接口和桥接
   - tc netem 和 tbf 的使用
   - 理解 BDP 并配置缓冲区
2. **如何理解拥塞控制**
   - 基于丢包 vs 基于速率
   - 窗口控制 vs Pacing

核心价值

这些知识和工具对任何网络性能相关的工作都至关重要，
其价值远超过单个算法的实现。

结语

网络拥塞控制是一个复杂而精妙的领域。
从 Reno、Cubic 到 BBR，每一代算法都在探索
吞吐量、延迟、公平性之间的平衡。

项目意义

通过实践深入理解了：

- 拥塞控制的核心机制
- 算法设计与实现的权衡
- 测试和调试的方法论

参考文献

- Cardwell, N., et al. (2016). *BBR: Congestion-Based Congestion Control*. ACM Queue. https://dl.acm.org/doi/10.1145/3009824
- Linux Kernel Source: `net/ipv4/tcp_bbr.c` https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/net/ipv4/tcp_bbr.c
- Quinn QUIC Implementation https://github.com/quinn-rs/quinn
- smoltcp Documentation https://docs.rs/smoltcp/

谢谢！

Questions?