计算机网络分层调研综述

—CS3611 课程论文：基于流水线可靠数据传输的实现与探究

潘文峥，何陈溟

摘要

计算机网络是将地理位置不同的具有独立功能的多台计算机及其外部设备，通过通信线路连接起来，在网络操作系统，网络管理软件及网络通信协议的管理和协调下，实现资源共享和信息传递的计算机系统。这篇综述论文作为计算机网络课程论文，全文结构分为两大板块：在**(一) 调研综述**中将基于 Kurose 和 Ross 的计算机网络教材中采纳的五级分层，依次对计算机网络中各层的核心内容进行梳理总结，并结合对技术前沿的调研概述其基本内容和发展脉络；在**(二) 案例研究**中，我们依照课程项目选题，利用 python 编程实现了基于流水线的可靠数据传输协议，对回退 N 步（GBN）和选择重传（SR）协议的细节和设计考量进行了探究；在此基础上，我们还编程实现了一个简易的 TCP 协议，在关注其丰富的控制机制的同时对出现的现象和可能存在的问题加以模拟和评估。**值得指出，从题目设计到项目完成的过程中，团队目标明确、规划得当、分工合理、协作顺利、任务量充实**。

Abstract

Computer networking is a computer system that connects geographically different multiple computers and their external equipment, mainly through communication lines, under which resources can be shared and coordinated and information can be transmitted. This review paper, as a course paper for computer networking, is well structured and divided into two general sections: **(1)** a research overview based on our textbook, which adopted the five-layer structure. We have summarized the core content of each layer in track of its development combined with the research on the technological frontier; **(2)** in the part of case study, we selected the topic according to the course project and implemented reliable data transmission based on pipeline by using Python programmes. The details and designing concerns of GBN and SR protocols are both studied and leveraged. Beyond that, We have also programmed a simple TCP protocol, concentrating on its rich mechanisms, during which possible circumstances and problems are well simulated and evaluated. **Equally worth mentioning, from the beginning of project design to the process of implementation, this team always works with clear goals, appropriate plans, pleasant cooperation as well as substantial task load.**

目录

1 计算机网络发展调研综述

1.1 应用层 (Application layer)

1.1.1 概述与核心内容

应用层是一个抽象层，它指定通信网络中主机使用的共享通信协议和接口方法。[1] 在 Internet 协议套件 (TCP/IP) 和 OSI 模型中都指定了应用层抽象。虽然两个模型对各自的最高层使用相同的术语，但详细的定义和目的是不同的。[2] 本文遵照 Kurose 书中采用的五层模型进行分析和梳理。

应用层是网络应用程序及它们的应用层协议存留的地方。因特网的应用层包括许多协议，例如 HTTP（提供 Web 文档的请求和传送）、SMTP（提供电子邮件报文的传输）和 FTP（提供两个端系统之间的文件传送) 等。此外，完成域名向 32 比特网络地址转换的域名系统（DNS）也是基于应用层的特定协议。应用层协议分布在多个端系统上，而一个端系统中的应用程序使用协议与另一个端系统中的应用程序交换信息分组，这种位于应用层的信息分组称为报文 (message)。[3] 这种将应用软件限制在端系统的方法，促进了大量网络应用程序的迅速研发和部署。

Web 和 HTTP Web 的应用层协议是超文本传输协议 (HyperText Transfer Protocol，HTTP)，它是 Web 的核心，由两个程序实现：一个客户程序和一个服务器程序。客户程序和服务器程序运行在不同的端系统中，通过交换 HTTP 报文进行会话。HTTP 定义了这些报文的结构以及客户和服务器进行报文交换的方式。HTTP 定义了 wcb 客户向 web 服务器请求 web 页面的方式，以及服务器向客户传送 web 页面的方式。

HTTP 使用 TCP 连接作为它的支撑运输协议，HTTP 是一个无状态协议，其并不保存关于客户的任何信息，分持续和非持续连接两种，主要区别在于每个 TCP 连接是否只传输一个请求报文和一个响应报文；一个典型的 HTTP 报文由状态行、首部行、空行和实体体构成。

Cookie 与 Web 缓存 尽管 HTTP 是无状态的，但出于 Web 站点对用户身份识别记忆的需要，HTTP 使用了 cookie。当用户首次浏览站点时，浏览器会对其 HTTP 报文中 Set-cookie 填入识别码，在此后该用户发往该站点的每个 HTTP 请求报文都包括该 cookie。尽管 cookie 的使用常常能够简化用户的网络活动中的操作，但也因为隐私边界问题受到广泛争议。

Web 缓存器又称代理服务器，是能够代表初始 Web 服务器满足 HTTP 的网络实体。部署了代理服务器以后，当浏览器请求某对象时，将先创建一个到 Web 缓存器的 TCP 连接，并向 Web 缓存器中的对象发送一个 HTTP 请求。若 Web 缓存器的本地存储中有该对象副本，就向客户浏览器用 HTTP 响应报文返回该对象，否则打开一个与该对象的初始服务器的 TCP 连接。Web 缓存器会在本地缓存一份副本，并向客户的浏览器用 HTTP 响应报文发送该副本。使用代理服务器的一大突出优点在于大幅降低网络时延，相较于提高链路速率而言，这种方案代价更小，得

到了广泛的应用。

SMTP 与电子邮件 电子邮件系统由用户代理、邮件服务器和简单传输协议（SMTP）组成。当用户 Alice 希望发送邮件给 Bob 时，首先调用她的邮件代理程序撰写报文并通过邮件地址发送报文，用户代理将其报文发送给她的邮件服务器（放入报文队列中）。当邮件服务器上的 SMTP 客户端发现该报文，就会通过 TCP 连接到 Bob 的邮件服务器上，并把该报文放入 Bob 的用户邮箱中，Bob 可以通过用户代理阅读并回复该邮件。由于 SMTP 只是一个“推”协议，目前流行的 POP3 协议和 IMAP 协议作为邮件访问协议解决了用户从其 ISP 的邮件服务器上获取邮件的操作问题。

今天，基于 Web 的电子邮件逐渐流行。浏览器作为用户代理，通过 HTTP 而非 SMTP 发送邮件到邮箱服务器，也通过 HTTP 从远程邮箱取得邮件。但是，邮箱服务器之间的通信仍由 SMTP 完成。

DNS 域名系统 DNS 是一个由分层的 DNS 服务器实现的分布式数据库，其作用是提供从主机名到 IP 地址的转换目录服务。分布式的 DNS 服务器分为三种类型，包括根 DNS 服务器、顶级域 DNS 服务器和权威 DNS 服务器。通过递归查询、迭代查询等方式，客户可以通过 DNS 报文的传递查询到相应的 IP 地址，而 DNS 缓存的存在能有效改善时延并减少在互联网上到处传输的 DNS 报文数量，它允许本地 DNS 服务器缓存最近查询的一些站点 IP，从而绕过大量的高级服务器的查询。

1.1.2 前沿发展

基于拓扑感知的流媒体编码和应用层组播系统设计 互联网的数据传播方式分为单播、组播和广播三种。[4] 在这三种互联网数据传播方式中，单播能够及时响应服务器和客户之间的请求，为网络用户提供个性化服务，但是单播传播方式却只能承载较小的流量；广播传播方式所需要的网络设备简单，对于信息的传输所需带宽较小，可以承受较大的流量，但是却无法为网络用户提供个性化服务，且较为容易耗尽带宽。基于单播和广播的优缺点，设计组播系统既可以承受较大的流量，又可以为网络用户提供个性化服务。目前，应用层组播系统设计已经成为互联网研究中的热点话题之一。[5] 一种创新提出的基于拓扑感知的流媒体编码和应用层组播系统设计，降低组播系统求解时间，提高组播系统收敛速度。

这一协议的主要内容：基于此次设立的数据运输处理流程，可以发现在数据传输过程中，总是以第一个运行的节点作为系统运行的启动节点。此时通过该节点可以向下一节点传输数据包，且一旦有其他节点的加入，必须向前一个节点传输连接请求，输入与启动节点相对应的 IP 和端口号 [6] 。因此可以以最先传递数据信息的节点创建一个组播组，此时的节点只有加入组播组，才可以正常接收数据包信息，并判断节点所加入的组播组是否正确，一旦发现节点所加入的组播组存在错误，会将节点及其传送的数据打包重新发回待加入节点。在组播组中有节点的加入

就会有节点的退出，节点退出组播组，其一因为当前节点没有子节点，数据包不能向下一节点进行传递；其二，为保证子节点获取的数据包信息是有效信息，当前节点会主动退出组播组。

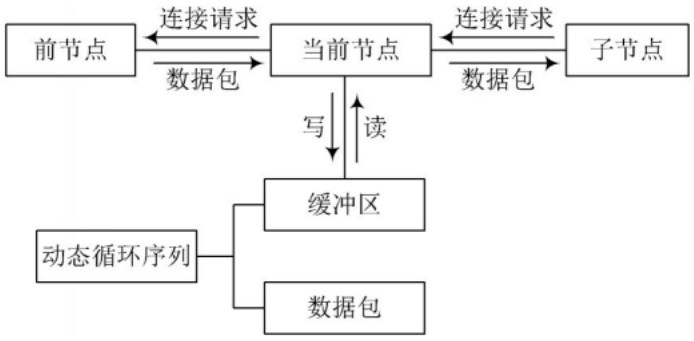


图 1: 数据运输处理流程

实验结果表明，这样设计的基于拓扑感知的组播系统设计与对照组传递速率基本完全一致，数据传递速率快且稳，且在传递过程中，带宽资源充裕。

基于集成学习的多类型应用层 DDoS 攻击检测方法 分布式拒绝服务攻击可操作性强、攻击门槛低，给网络服务商带来客户流失、商业损失等重大风险。与传统基于低层协议的 DDoS 攻击相比，应用层 DDoS 攻击利用高层协议实现，模拟正常用户的访问行为，具有难以检测的标志性特征 [7]。目前较为流行的低层检测系统难以判断用户请求，导致检测误报率高等问题。现有 DDoS 攻击检测方法采用传统的基于统计的或是基于机器学习的检测方法，只区分异常流量和正常流量，未进一步检测流量中攻击的具体类型，且检测准确率低。为解决现有技术存在的问题，一种创新检测多类型应用层 DDoS 攻击的检测方法，检测多类型应用层 DDoS 攻击有助于后续针对性的防御措施。该方法采用基于集成学习的多分类检测模型，能够检测多类型的应用层 DDoS 攻击，包括挑战黑洞（Challenge Collapsar，CC）、HTTP Flood、HTTP Post 及 HTTP Get 攻击共四种应用层 DDoS 攻击。

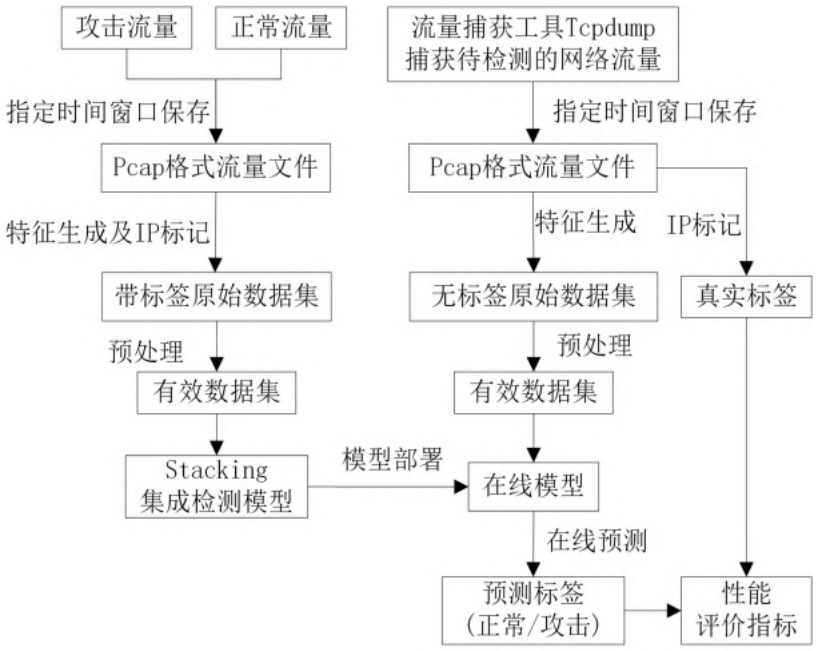


图 2: 检测算法流程

基于训练好的模型检测实时应用层 DDoS 攻击，在线检测模块通过在线部署的检测模型判断待检测流量，利用 Shell 脚本实现在线流量数据采集、在线数据集生成、在线检测的自动化过程。在线流量数据采集阶段，利用实时流量抓取工具，在网络流量入口处间隔指定时间捕获流经流量。在线数据集生成阶段，在线读取生成的流量文件，采用流特征提取工具 CICFlowMeter 将指定时间间隔在线保存的流量文件转为包含 87 维特征信息的数据集，根据数据预处理模块的特征工程方法提取数据集的 47 维有效特征，生成在线检测数据集。在线检测阶段，部署在网络流量入口处的检测模型作为在线分类器，判断输入流量的类型，生成模型预测标签。通过流量真实标签以及模型预测标签，计算准确率、误报率、召回率以及恶意流量检测率等指标，验证模型在线检测性能。

该方法提出的基于集成学习的应用层 DDoS 攻击检测方法，结合离线训练和在线检测方法，可检测多类型应用层 DDoS 攻击。实验表明，可有效检测多类型应用层 DDoS 击，在最优时间窗口下恶意流量检测率达 98%。进一步验证基于集成学习的检测方法在真实网络环境中的可用性，优化在线时间窗口值并优化模型，得到更佳检测效果。[8]

1.2 传输层 (Transport layer)

1.2.1 概述与核心内容

因特网的运输层在应用程序端点之间传送应用层报文。在因特网中，有两种运输协议，即 TCP 和 UDP，利用其中的任一个都能运输应用层报文。TCP 向它的应用程序提供了面向连接

的服务。这种服务包括了应用层报文向目的地的确保传递和流量控制（即发送方/接收方速率匹配）。TCP 也将长报文划分为短报文，并提供拥塞控制机制，因此当网络拥塞时，源抑制其传输速率。UDP 协议向它的应用程序提供无连接服务。这是一种不提供不必要服务的服务，没有可靠性，没有流量控制，也没有拥塞控制。在本书中，我们把运输层的分组称为报文段 (segment)。

无连接运输：UDP UDP 只做运输协议能够做的最少工作，从应用进程得到数据，附加上用于多路复用/分解服务的源和目的端口号字段，以及两个其他的小字段，然后将形成的报文段交给网络层。网络层将该运输层报文段封装到一个 IP 数据报中，然后尽力而为地尝试将此报文段交付给按收主机。如果该报文段到达接收主机，UDP 使用目的端口号将报文段中的数据交付给正确的应用进程。值得注意的是，使用 UDP 时，在发送报文段之前，发送方和接收方的运输层实体之间没有握手。正因为如此，UDP 被称为是无连接的。UDP 的报文段十分简洁，由 32 比特的源端口号、目的端口号、长度和校验和以及应用数据构成，校验和虽然提供基础的差错检验，但对差错恢复无能为力。

流水线可靠传输协议 通过每次的迭代和使用场景越来越接近实际情况，我们可以构造 rdt 从完全可信信道的 1.0 版本到具有比特差错的丢包信道的可靠数据传输（rdt3.0），序号、冗余分组和倒计时定时器等的引入使得传输过程中出现的错误和丢包问题得以克服。在停等协议的基础上，为了提高信道的利用率，两种流水线可靠数据传输协议应运而生。这也是本文第二个板块的大作业实践部分重点关注的选题。回退 N 步协议中，允许发送方发送多个分组而不需等待确认；而选择重传协议中，为了解决 GBN 中一个分组的错误会引起大量不必要的重传的性能浪费，选择重传允许发送方仅发送那些可能出错的分组，同时缓存所有失序分组。

这部分内容是本文案例分析板块作为重点分析的对象，我们将用 python 程序实现 GBN 和 SR 两个协议，并探讨其设计思想和不同参数改变带来的传输影响。

TCP 和拥塞控制 TCP 是一个面向连接的传输协议，因为在一个应用进程向另一个应用进程发送数据之前，两个进程必须先经过“三次握手”机制确保建立可靠连接。首先由客户端 TCP 向服务器端发送一个特殊的 TCP 报文段，其首部 SYN 比特（标志位）被置 1，并随机化选择一个 client_isn 发送；第二步，当包含 SYN 的报文段到达服务器端，服务器会读取 SYN 和客户序号并分配缓存和变量，再回复 SYNACK 报文段；最后，客户收到 ACK 后，将发送一个 SYN 置 0 的确认报文段完成连接。同时，TCP 为它的用户提供了流量控制和拥塞控制服务，前者通过发送方维护的接收窗口变量实现，而后者则基于慢启动、拥塞避免和快速恢复三个状态的交替实现。

在本文案例分析的拓展实践部分，我们将编程实现一个建议的 TCP 协议，并尽可能支持其丰富的控制特性，并在此基础上探讨其连接管理、流量控制、拥塞控制等技术细节。

1.2.2 前沿发展

认知无线电网络中一种改进的传输层协议 认知无线电无线网络由 2 种用户组成：主用户和认知用户。认知用户以“机会主义”的方式感知空闲频段，并利用该频段传输数据。一旦发现主用户的通信活动就停止传输，重新感知并切换到其他空闲频段 [9]。如何提高认知无线电网络的传输性能是一个重要的研究方向，其中 TCP 的性能是影响认知无线电网络传输性能的重要因素。

传统的 TCP 协议是针对有线网络设计的，而无线网络中由于无线链路质量低等因素导致传输层频繁重传，使 TCP 协议的性能大大降低。针对以上问题，研究者创新提出了 TCP-Vegas 和 TCP-Westwood 等协议。随着认知无线电技术的引入，这些 TCP 协议并不能很好地适应认知无线电网络的特性。因此，在 TCP-Reno 协议的基础上提出了适用于认知无线电网络的协议——TCP-Cog。该协议进行了传输预判及信道切换后 SSThresh 自适应调整的改进，提高了传输层的整体性能。

TCP-Cog 协议采用了跨层设计思想，TCP-Cog 在收到 MAC 层感知通知后，将应用层不断下发的数据包以队列的形式暂存。在感知结束后，检测数据包队列，如果不为空则将队列里的数据按入队的顺序发送。这样有效解决了感知阶段数据包溢出丢弃问题，避免了丢包重传和窗口减少。同时 SSThresh 的自适应调整也用于改善传输效率问题。TCP-Cog 中，频谱感知结束时，MAC 层会将感知的结果传递给传输层。如果当前信道被主用户占用，而认知用户又没有探测到其他空闲频段，感知定时器会将超时时间延长一个感知周期。在此期间，TCP 的传输是暂停的。SSThresh、cwnd 和 RTT 等参数也被暂存了下来。如果之后感知到的空闲信道仍为先前的那条信道，则利用暂存的参数恢复传输；当且仅当切换了传输信道时，TCP-Cog 将 cwnd 置为一，并预测切换信道的带宽。

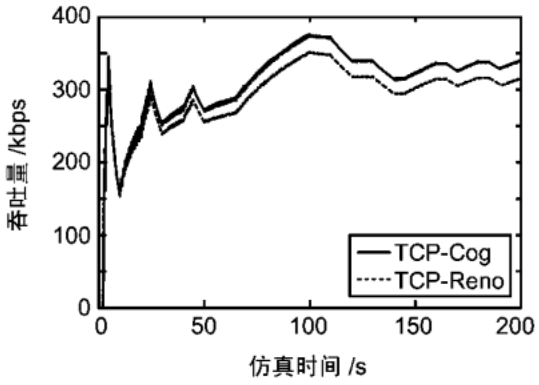


图 3: TCP-Cog 拥塞窗口与慢启动门限随时间的变化

上述研究全面分析了认知无线电网络中 TCP 遇到的新问题，并运用跨层设计思想对传统 TCP 协议进行了传输预判与慢启动门限值自适应改进。仿真结果证明，TCP-Cog 显著减小了重传数据包、提高了实际传输速率、改善了 TCP 在认知无线电网络中的性能。[10]

1.3 网络层 (Network layer)

1.3.1 概述与核心内容

网络层是计算机网络中至关重要的层次结构,因特网的网络层负责将称为数据报(datagram)的网络层分组从一合主机移动到另台主机。在一台源主机中的因特网运输层协议（TCP 或 UDP）向网络层递交运输层报文段和目的地址。因特网的网络层包括著名的网际协议（IP），该协议定义了在数据报中的各个字段以及端系统和路由器如何作用于这些字段。IP 仅有一个，所有具有网络层的因特网组件必须运行 IP。因特网的网络层也包括决定路由的路由选择协议，它根据该路由将数据报从源传输到目的地，为了叙述的简洁，我们不再将网络层的数据平面和控制平面独立分节。

路由器和分组调度 路由器的主要职责可以概括为转发和路由选择两个部分。其中转发是数据平面的本地动作，通常时间较短；路由选择则是确定从源到目的地所采取的端到端路径网络范围的处理过程。每台路由器中有一个关键元素，即转发表，路由器将检查到达分组的首部值并在其转发表中检索以确定其输出的链路端口。对于控制平面的路由选择算法，既有传统的由路由器厂商在其产品中采用直接配置的方法，也可以通过软件自定义网络（SDN）实现控制平面和数据平面的分离，允许 ISO 去创新更改控制平面的软件功能。

交换结构处于路由器的核心，通常有经内存交换、经总线交换和经互联网络交换三种形式，在输入输出端口都可能出现丢包和排队，分组调度至关重要。先进先出、优先权排队、循环和加权公平排队在不同的应用场景为分组的链路传输次序做出了规定。

IP 编址、CIDR 和 NAT IPv4 数据报中包含版本号、首部长度、服务类型、数据报长度、标识、标志、位偏移、寿命、协议、首部校验和、源和目的 IP 地址、选项和有效载荷（数据）等关键字段。我们重点关注 IPv4 的编址方式，IP 要求每台主机和路由器接口拥有自己的 IP 地址。每个 IP 地址长度为 32 比特，采用点分十进制书写，一个 IP 地址的一部分需要由其子网决定。例如：223.1.1.0/24 中的/24 称为子网掩码，要求任何连接到该网络的主机都具有 223.1.1.xxx 格式。

考虑到具有 8、16 和 24 比特子网地址的 A、B、C 类子网的大小并不能合适地满足需求，一种称为无类别域间路由选择的方案（CIDR）应运而生，它将子网的概念一般化了，用 $a.b.c.d/x$ 表示子网，其中 x 表示第一部分中的比特数。

网络地址转换（NAT）为子网范围的扩充提供了解决方案。用于某局域范围的专用网络分布在 NAT 一侧，可以用自由的 IP 编址，并利用 NAT 转换表实现 IP 和端口号的转换。因此，NAT 对外的行为看上去是具有单一 IP 地址的单一设备，这也为特定的安全、隐私需求提供了保障思路。

路由选择算法 两种广泛使用的路由选择算法是链路状态算法和距离向量算法。

链路状态算法的本质是 Dijkstra 算法，其理论依据是根据贪心算法寻找单源最短路径：首先将起点的距离设为 0，其余点距离为无穷大，每次迭代选取到源点距离最小的节点加入已确定的节点集合，并逐一检查其邻接点是否可以通过该节点更新距离，直到全部节点的最短距离被确定。

距离向量算法的实质是 Bellman-Ford 算法，由著名的 Bellman-Ford 方程给出其核心步骤：

$$d_x(y) = \min_v\{c(x,v) + d_v(y)\}$$

其中 $d_x(y)$ 是节点 x 到节点 y 的最低开销路径的开销，$c(x,v)$ 是 x,v 边的开销。需要注意的是，距离向量算法和链路状态算法最大的不同就在于它是分布式的、异步的。每个节点对其每个邻居执行上述方程的更新操作，并持续向其邻居发送他的距离向量副本。

OSPF 和 BGP 因特网是 ISP 的网络，通过路由器组织进自治系统可以解决因规模扩大造成的管理自治需求。开放最短路优先 (OSPF) 协议用于因特网的 AS（自治系统）内部路由选择。OSPF 的鲜明特征是它是一种链路状态算法，它使用 dijkstra 最短路算法和泛洪链路状态信息。其优点主要体现在安全（能够鉴别 OSPF 路由交换防止恶意入侵）、允许多条相同开销路径选择（避免单一路径承载过多流量）以及对单播和多播的综合支持，并且支持在单个 AS 中的层次结构。

与 OSPF 互为补充的重要协议是 BGP（边界网关协议）作为自治系统间的路由选择协议，将因特网中的 ISP 黏合起来，它的主要作用是从邻居 AS 获得前缀的可达性信息并确定到该前缀的最佳路由。BGP 分为 eBGP 和 iBGP，分别负责跨越 AS 的外部链接和 AS 内部的连接，其路由选择算法是热土豆路由选择基础上的距离向量算法。

1.3.2 前沿发展

基于回溯的 QKD 网络随机路由选择算法 我们知道 RIP 协议的随机路由算法是通过改进基于距离矢量的路由算法扩展路由表 [11]，为到达某一目的地址的中继节点添加多个下一跳，进而得到源中继节点到目的中继节点的多条最短路径；在选路时，将链路上的剩余密钥量考虑在内，在存在多个密钥量充足的下一跳中继节点的情况下，在其中随机地选择一个，然后逐跳转发，直到转发至目的中继节点。虽然该方法路径最短且安全性较高，但仍然存在以下不足：1) 只适用于源节点和目的节点之间有多条最短路径的情况，如果源、目的节点之间只有一条最短路径，该算法就会失效；2) 在选中密钥量不足的下一跳节点时就将中断传输，按新的最短路径从头开始密钥协商，浪费资源且成功率较低。针对该随机路由算法的不足，有研究者创新性地将该路由算法进行改进，并提出一种基于回溯的 QKD 网络随机路由选择方案。

这种改进算法的基本思想是：从源节点开始进行寻路，在当前节点与目的节点存在多条传输路径时，查看链路上的剩余密钥量，在剩余密钥量足够的所有链路中随机选择一条，逐跳选择中继节点来转发密钥。当选择的节点不存在密钥量足够的链路时，判断该节点是否存在可回溯的中继节点。如果存在，就回溯；如果不存在，结束通信，重新开始协商密钥。为了找到可

回溯的中继节点，研究者设计了一种标记回溯点的方法。在路由表中添加一个新项——回溯点，用于标记该节点的回溯点。即，如果所选路径中某节点的链路密钥量不足，此节点就查找其回溯点。如果存在，就回溯到该点；如果不存在，就结束此次密钥协商。标记回溯点的算法步骤如下：1) 初始情况下，路由表中每一项的回溯点均为空；2) 从源中继节点开始，判断当前节点到目的节点的下一跳个数：如果个数为 1，将下一跳的回溯点标记为当前节点的回溯点的值；如果个数大于 1，将多个下一跳的回溯点标记为当前节点。图 3 是一个简单的例子，如果 Alice 要与 Bob 通信，计算得到 3 条最短路径：P1：1→2→3→4；P2：1→2→5→4；P3：1→8→6→4。所以，节点 2,8 的回溯点为 1；节点 3,5 的回溯点为 2。

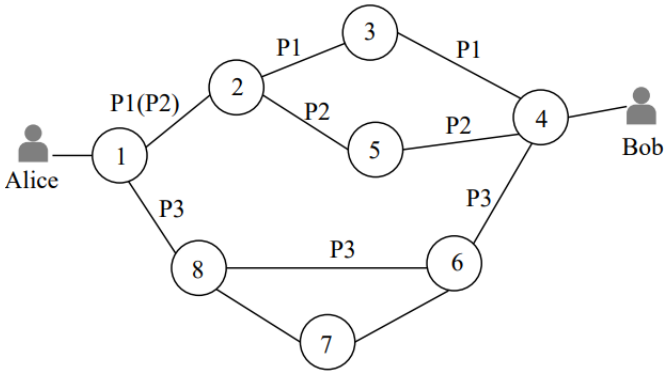


图 4: 标记回溯点的例子

这种改进的路由选择算法主要针对基于信任中继的量子密钥网络的路由问题展开研究，分析了现有路由方案存在的问题，提出了一种随机路由方案。该方案通过改进的路由算法找到两点之间的 3 条最短路径，从而保证有多条较短路径；在选路过程中，随机选择其中的一条，安全性较高；遇到密钥量不足的链路时，回溯到有其他路径的节点，节省密钥量并提高了密钥传输效率。实验及分析结果表明，本算法不仅适用范围广，并且在选路时间、密钥消耗量以及密钥传输效率方面有一定的优势。[12]

1.4 链路层 (Link layer)

1.4.1 概述与核心内容

运行链路层协议的任何设备均称为**节点 (node)**。节点包括主机、路由器、交换机和 WiFi 接入点。沿着通信路径链接相邻节点的通信信道被称为**链路 (link)**。在通过特定的链路时，传输节点将数据封装在**链路层帧**中，并将该帧传送到链路中。链路层提供的服务包括**成帧**、**链路接入**、**可靠交付和差错检测和纠正**。链路层帧的主体部分是在**网络适配器**中实现的。

差错检测和纠正技术 发送方使用**差错检测和纠正比特 (EDC)** 来增强数据 D。接收方接受到比特序列 D’ 和 EDC’。接收方的挑战是在只收到 D’ 和 EDC’ 的情况下，确定 D’ 是否和初始的 D

相同。有三种检测差错的技术: **奇偶校验位 (parity bit)**、**检验和方法**和**循环冗余检测**。

多路访问链路和协议 **点对点链路**由链路一端的单个发送方和链路另一端的单个接收方组成。**广播链路**能够让多个发送和接收节点都连接到相同的、单一的、共享的广播信道上。任何多路访问协议能被划分为 3 种类型之一: **信道划分协议，随机接入协议**和**轮流协议**。

交换局域网 交换机运行在链路层，交换链路层帧。主机和路由器同时具有网络层和链路层地址。

地址解析协议 (ARP) 提供了将 IP 地址转换为链路 7 层地址的机制。

MAC 地址: 即通常意义下的链路层地址。MAC 地址长度为 6 字节。MAC 地址具有扁平结构，无论适配器到哪都不会变换。

地址解析协议 (ARP): 发送主机的 ARP 取相同主机下的 IP 地址作为输入返回对应的 MAC 地址。ARP 只为**同一子网**上的主机和路由器接口解析 IP 地址。交换局域网还包括**以太网，链路层交换机和虚拟局域网技术**。

链路虚拟化 将网络作为链路层的技术。

数据中心 数据中心网络将内部主机彼此互联并与因特网中的数据中心互联。在数据中心内部，外部请求首先被定向到一个**负载均衡器**。当主机规模扩大，数据中心通常应用**路由器和交换机等级结构**。

1.4.2 前沿发展

近年来，无线链路层，或虚拟链路层有较大发展。大规模通信网络仿真环境是信息系统设计、开发和测试的重要基础。基于链路层虚拟化技术可根据信息系统研制过程中对通信网络环境的应用需求制定通信网络仿真技术。为解决虚拟链路数量容纳比较少、实时性差的问题，有许多新技术在被广泛地研究，如 AFDX 端系统。

1.5 物理层 (Physical layer)

1.5.1 概述与核心内容

传输介质 传输介质主要分为**双绞线, 同轴电缆，光缆，无线传输**和**卫星通信**。介质依据传输速率、传输距离、稳定性等不同特性被用于不同的场景。

比特与信号 数字数据最终都被表示成比特串，数字传输的本质就是把比特串变成信号。任何信号的传输都可理解为以**傅里叶级数**的形式传递。数据传输以信号为载体，可分为**模拟传输**和

数字传输。有三种将数字数据调制成模拟信号进行传输的基本方式: **调幅 ASK**、**调频 FSK** 和**调相 PSK**。常用的对数字信号编码的 **不归零编码**、**曼切斯特编码**、**差分曼切斯特编码**和 **4B/5B 编码**。此外较为重要的定理有 **Nyquist 定理**和**香农定理**。

多路复用 多路复用主要有两种: **频分多路复用 FDM** 和**时分多路复用 TDM**。

传输模式 有 3 对传输模式。**串行传输**和**并行传输**、**基带传输**和**宽带传输**、**异步传输**和**同步传输**。

物理层的接口 物理层的协议体现在物理接口上。物理接口主要分为**异步串行口 RS-232**、**常用的并行口**和 **RJ45**。

电话系统 PSTN 通常的计算机通过 MODEM 拨号上网，即电话系统传输数据。电话系统一般为树状结构

Internet 接入 Internet 的本地接入可分为**拨号接入**、**ADSL 接入**和 **Internet over Cable(基于社区电视系统)**。

1.5.2 前沿发展

以 5G(NR) 技术为例。5G 技术的物理层的协议是 38.2xx 系列。NR 与 LTE 系统都基于 OFDM 传输。两者主要有两点不同：

(1) LTE 只支持一种子载波间隔 15KHz，而 NR 目前支持 5 种子载波间隔配置;

(2) LTE 上行采用基于 DFT 预编码的 CP-Based OFDM，而 NR 上行可以采用基于 DFT 预编码的 CP-Based OFDM，也可以采用不带 DFT 的 CP-Based OFDM。

5G 物理层中还应用了许多关键技术，如：MIMO 技术，多天线技术，低密度奇偶校验 (Low Density Parity Check，LDPC) 码等。

2 案例研究

2.1 基础任务：流水线可靠传输协议

2.1.1 回退 N 步 (GBN)

实现效果

我们在接收端设置了错误和丢包模拟器，当丢包概率和错误概率均设置为 0.1，窗口长度 $N = 4$，序号空间 $L = 8$，输入为 "$abcdefghijk$" 时，一次典型的传输过程如下所示。

Sender 端：

GBN 协议 sender 端实例化：

Input sentence:

abcdefghijk

Packege0 MessagePkt: a has been sent

Packege1 MessagePkt: b has been sent

Packege2 MessagePkt: c has been sent

Packege3 MessagePkt: d has been sent

Receive: 'pktID': -1, 'data': 'ACK', 'checksum': True

Receive: 'pktID': -1, 'data': 'ACK', 'checksum': True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege0 MessagePkt: a has been sent

Packege1 MessagePkt: b has been sent

Packege2 MessagePkt: c has been sent

Packege3 MessagePkt: d has been sent

Receive: 'pktID': 0, 'data': 'ACK', 'checksum': True

Packege4 MessagePkt: e has been sent

Receive: 'pktID': 1, 'data': 'ACK', 'checksum': True

Packege5 MessagePkt: f has been sent

Receive: 'pktID': 1, 'data': 'ACK', 'checksum': True

Receive: 'pktID': 1, 'data': 'ACK', 'checksum': True

Receive: 'pktID': 1, 'data': 'ACK', 'checksum': True

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege2 MessagePkt: c has been sent

Packege3 MessagePkt: d has been sent

Packege4 MessagePkt: e has been sent

Packege5 MessagePkt: f has been sent

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege2 MessagePkt: c has been sent

Packege3 MessagePkt: d has been sent

Packege4 MessagePkt: e has been sent

Packege5 MessagePkt: f has been sent

Receive: ’pktID’: 2, ’data’: ’ACK’, ’checksum’: True

Packege6 MessagePkt: g has been sent

Receive: ’pktID’: 3, ’data’: ’ACK’, ’checksum’: True

Packege7 MessagePkt: h has been sent

Receive: ’pktID’: 4, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 5, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 6, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 7, ’data’: ’ACK’, ’checksum’: True

Packege0 MessagePkt: i has been sent

Packege1 MessagePkt: j has been sent

Packege2 MessagePkt: k has been sent

Packege3 MessagePkt:

has been sent

Receive: ’pktID’: 0, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 2, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 3, ’data’: ’ACK’, ’checksum’: True

Finish

Receiver 端：

GBN 协议 receiver 端实例化：

Package lost.

’pktID’: 1, ’data’: ’b’, ’checksum’: False

Package lost.

’pktID’: 3, ’data’: ’d’, ’checksum’: True

’pktID’: 0, ’data’: ’a’, ’checksum’: True

strBuffer: A

’pktID’: 1, ’data’: ’b’, ’checksum’: True

strBuffer: AB

’pktID’: 2, ’data’: ’c’, ’checksum’: False

’pktID’: 3, ’data’: ’d’, ’checksum’: True

’pktID’: 4, ’data’: ’e’, ’checksum’: True

’pktID’: 5, ’data’: ’f’, ’checksum’: True

Package lost.

’pktID’: 3, ’data’: ’d’, ’checksum’: True

’pktID’: 4, ’data’: ’e’, ’checksum’: True

’pktID’: 5, ’data’: ’f’, ’checksum’: True

’pktID’: 2, ’data’: ’c’, ’checksum’: True

strBuffer: ABC

’pktID’: 3, ’data’: ’d’, ’checksum’: True

strBuffer: ABCD

’pktID’: 4, ’data’: ’e’, ’checksum’: True

strBuffer: ABCDE

’pktID’: 5, ’data’: ’f’, ’checksum’: True

strBuffer: ABCDEF

’pktID’: 6, ’data’: ’g’, ’checksum’: True

strBuffer: ABCDEFG

’pktID’: 7, ’data’: ’h’, ’checksum’: True

strBuffer: ABCDEFGH

'pktID': 0, 'data': 'i', 'checksum': True
strBuffer: ABCDEFGHI
'pktID': 1, 'data': 'j', 'checksum': True
strBuffer: ABCDEFGHIJ
'pktID': 2, 'data': 'k', 'checksum': True
strBuffer: ABCDEFGHIJK
'pktID': 3, 'data': 'newline', 'checksum': True
strBuffer: ABCDEFGHIJK
The received messages through GBN:
ABCDEFGHIJK

容易看到，当错误发生时，接收方发送上一个按需接受的分组序号的 ACK，当发送方最早发送的分组超时后，将执行重新发送窗口内所有未确认分组的操作，直到按序 ACK 被收到，则前移窗口。正确实现了 GBN 协议的动作。

问题回答

1. GBN 协议相对于停等协议的改进在于？这一改进起到优化作用的原因为？

停等协议的主要不足在于发送方的信道利用率非常低，而 GBN 协议大幅提高了发送方利用率。优化作用的原因是流水线操作允许发送方发送多个分组而无需等待确认，大大提高了传输性能。

2. GBN 协议中发送方接受到 ACK 的响应是什么？主要特点是什么，请描述这一过程。

发送方收到一个 ACK 后，首先将执行滑动窗口操作，即把窗口的最小序号（base）前移到 ACK 的序号加一；接着，如果仍有已发送但未确认的分组，则重启定时器，如果全部分组已确认，则停止计时器。

3. GBN 协议接收方接受到正确的分组时的响应是什么？其他情况？

当接收方收到正确的分组时，将发送一个具有和所收到分组相同序号的 ACK 分组；若收到的分组失序，接收方仍然发送一个 ACK，但序号值为上一个正确按序收到的分组的序号。

4. GBN 协议发送方启用的计时器数量为？超时事件的响应为？

GBN 协议始终只使用一个计时器，即维护最早已发送但未确认的分组的时间。当超时事件发生时，发送方重新发送窗口中所有已发送但未确认的分组。

5. 为什么可用序号的长度要设计为窗口长度的两倍，如果相等会怎样？考虑并描述会引发冲突的情况。

如果窗口长度与序号相等，如 $L = N = 4$，考虑以下情形：发送方第一轮发送的四个分组记为 A_0, A_1, A_2, A_3，接收方正确接收了这四个分组并发回相应的 ACK，但是这四个 ACK 全部丢包。此后，发送方由于没有受到任何一个正确的 ACK，超时后将重发一遍 A_0, A_1, A_2, A_3，由于这次重发的分组恰好按序具有接收方期待的序号 (0,1,2,3)，接收方将直接将这四个分组接收，但事实上，这些分组并不是真正的第二轮分组，从而产生错误。因此，窗口大小应严格小于序号空间大小（即 $N \leq L - 1$）以避免出现上述情形，通常，我们取 $N = \frac{L}{2}$。

通过对 GBN 协议代码的修改，当窗口和序号长度均为 4 时作如下测试：

GBN 协议 sender 端实例化：

Input sentence:

aaaabbbb

Packege0 MessagePkt: a has been sent

Packege1 MessagePkt: a has been sent

Packege2 MessagePkt: a has been sent

Packege3 MessagePkt: a has been sent

Receive: None

Package lost.

Receive: None

Package lost.

Receive: None

Package lost.

Receive: None

Package lost.

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege0 MessagePkt: a has been sent

Packege1 MessagePkt: a has been sent

Packege2 MessagePkt: a has been sent

Packege3 MessagePkt: a has been sent

GBN 协议 receiver 端实例化：

'pktID': 0, 'data': 'a', 'checksum': True

strBuffer: A

'pktID': 1, 'data': 'a', 'checksum': True

strBuffer: AA

'pktID': 2, 'data': 'a', 'checksum': True

strBuffer: AAA

'pktID': 3, 'data': 'a', 'checksum': True

strBuffer: AAAA

'pktID': 0, 'data': 'a', 'checksum': True

strBuffer: AAAAA

'pktID': 1, 'data': 'a', 'checksum': True

strBuffer: AAAAAA

'pktID': 2, 'data': 'a', 'checksum': True

strBuffer: AAAAAAA

'pktID': 3, 'data': 'a', 'checksum': True

strBuffer: AAAAAAAA

'pktID': 0, 'data': 'a', 'checksum': True

可以看到，输入的 $aaaabbbb$ 字符串被错误接收为 $AAAAAAAA$，模拟情形与预期一致.

2.1.2 选择重传 (SR)

实现效果

Sender 端：

SR 协议 sender 端实例化：

Input sentence:

qwertyuiopa

Packege0 MessagePkt:q has been sent

Packege1 MessagePkt:w has been sent

Packege2 MessagePkt:e has been sent

Packege3 MessagePkt:r has been sent

Receive: 'pktID': 0, 'data': 'ACK', 'checksum': True

Packege4 MessagePkt:t has been sent

Receive: 'pktID': 3, 'data': 'ACK', 'checksum': True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege1 MessagePkt:w has been sent

Timeout. Resend the package

Packege2 MessagePkt:e has been sent

Timeout. Resend the package

Packege4 MessagePkt:t has been sent

Receive: 'pktID': 2, 'data': 'ACK', 'checksum': True

Receive: 'pktID': 4, 'data': 'ACK', 'checksum': True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege1 MessagePkt:w has been sent

Waiting ACK...

Receive: 'pktID': 1, 'data': 'ACK', 'checksum': True

Packege5 MessagePkt:y has been sent

Packege6 MessagePkt:u has been sent

Packege7 MessagePkt:i has been sent

Packege0 MessagePkt:o has been sent

Receive: ’pktID’: 5, ’data’: ’ACK’, ’checksum’: True

Packege1 MessagePkt:p has been sent

Receive: ’pktID’: 7, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 0, ’data’: ’ACK’, ’checksum’: True

Receive: ’pktID’: 1, ’data’: ’ACK’, ’checksum’: True

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege6 MessagePkt:u has been sent

Waiting ACK...

Receive: ’pktID’: 6, ’data’: ’ACK’, ’checksum’: True

Packege2 MessagePkt:a has been sent

Packege3 MessagePkt:

has been sent

Waiting ACK...

Receive: ’pktID’: 3, ’data’: ’ACK’, ’checksum’: True

Waiting ACK...

Timeout. Resend the package

Packege2 MessagePkt:a has been sent

Waiting ACK...

Waiting ACK...

Timeout. Resend the package

Packege2 MessagePkt:a has been sent

Waiting ACK...

Receive: ’pktID’: 2, ’data’: ’ACK’, ’checksum’: True

Finish

Receiver 端：

SR 协议 receiver 端实例化：

’pktID’: 0, ’data’: ’q’, ’checksum’: True buffered.

packet 0 delivered.

’pktID’: 1, ’data’: ’w’, ’checksum’: False Package lost.

’pktID’: 3, ’data’: ’r’, ’checksum’: True buffered.

Package lost.

’pktID’: 1, ’data’: ’w’, ’checksum’: False

’pktID’: 2, ’data’: ’e’, ’checksum’: True buffered.

’pktID’: 4, ’data’: ’t’, ’checksum’: True buffered.

’pktID’: 1, ’data’: ’w’, ’checksum’: True buffered.

packet 1 delivered.

packet 2 delivered.

packet 3 delivered.

packet 4 delivered.

’pktID’: 5, ’data’: ’y’, ’checksum’: True buffered.

packet 5 delivered.

’pktID’: 6, ’data’: ’u’, ’checksum’: False

’pktID’: 7, ’data’: ’i’, ’checksum’: True buffered.

’pktID’: 0, ’data’: ’o’, ’checksum’: True buffered.

’pktID’: 1, ’data’: ’p’, ’checksum’: True buffered.

’pktID’: 6, ’data’: ’u’, ’checksum’: True buffered.

packet 6 delivered.

packet 7 delivered.

packet 0 delivered.

packet 1 delivered.

Package lost.

’pktID’: 3, ’data’: ”, ’checksum’: True buffered.

’pktID’: 2, ’data’: ’a’, ’checksum’: False

’pktID’: 2, ’data’: ’a’, ’checksum’: True buffered.

packet 2 delivered.

packet 3 delivered.

The received messages through GBN:

QWERTYUIOPA

问题回答

1. SR 协议相对于 GBN 协议的改进在于？这一改进起到优化作用的原因为？（考虑失序分组）

(1) SR 协议让发送方仅重传在接受方出错的分组，避免了不必要的重传

(2) 原因是缓存了失序分组直到所有丢失分组被接收到。

2. SR 协议中发送方接受到 ACK 的响应是什么？是否仍采用累计确认的方法？请描述这一过程。

(1) 接收到 ACK 后判断是否需要移动窗口。移动后窗口中有可发送的分组则发送，否则等待；

(2) 不采用累计确认；

(3) 对应失序分组，将其缓存，收到正确序号的分组后将其与所有缓存的分组一起交付。

3. SR 协议接收方接受到分组时的响应是什么？讨论几种情况。（窗口内分组，窗口外分组，损坏分组）

(1) 窗口内分组：序号等于基序号则连同所有缓存交付。否则，如果未被受到过将其缓存；

(2) 窗口外分组：序号在 $[rcv\_base - N, rcv\_base - 1]$ 内分组，必须产生一个 ACK。其他情况则忽略该分组；

(3) 损坏分组：不发送 ACK 并打印损坏原因 (错误或丢失)。

4. SR 协议发送方启用的计时器数量为？超时事件的响应为？

SR 发送方启用的计时器数目与窗口长度相同；重传分组并重置对应的计时器。

2.2 拓展探究：TCP 协议实现

作为拓展思考部分，我们将在以上实现的 rdt 协议的基础上，探讨并尽可能实现 TCP 协议的控制内容，并进行分析、模拟和评估，进而准确、直观地把握这一传输控制协议的实质和特点。以下部分展示的是我们实现或模拟分析的 TCP 协议机制，包括：**超时时间间隔**（涉及 EstimatedRTT 和 DevRTT 的计算）、**序号与确认号机制**、**可靠传输机制**（超时间隔加倍等）、**连接管理机制**（三次握手等）以及对**拥塞控制机制**的模拟探究。

2.2.1 超时间隔控制

我们知道，TCP 和前述 rdt 协议一样，采用超时/重传机制来处理报文段丢失问题，但超时时长间隔的设置是一个值得探讨的问题。我们将基于 GBN 协议实现的可靠数据传输来测得每一个分组从发出到正确的 ACK 被接收所经历的时间间隔作为 “样本 RTT”（SampleRTT）。在实际网络环境的传输中，由于路由器拥塞和端系统负载的变化，样本 RTT 的值将随时间发生较大的波动，因而 TCP 采用以下迭代式来更新 SampleRTT：

$$EstimatedRTT = (1-\alpha)\cdot EstimatedRTT + \alpha \cdot SampleRTT$$

在实验中，我们取 [RFC 6298] 中给出的 $\alpha = 0.125$ 进行测定。

除了样本 RTT 以外，我们还将对 RTT 偏差时间 (DevRTT) 进行测量，迭代式如下：

$$DevRTT = (1-\beta)\cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

RTT 偏差值反映的是样本 RTT 时间的波动程度，波动越大则偏差值越大，反之亦然。在实验中，我们取 β 的推荐值 0.25。

在每次收到正确的 ACK 分组后，我们都将迭代计算上述三个参数，并利用这些参数更新每一轮传输的超时时长间隔，以确保其不至过小而发送过多重复分组，也不至过大而浪费过长的等待时间，实质上，下式给出的动态更新方案基于样本 RTT 的指数加权移动均值和偏差量：

$$TimeoutInterval = EstimatedRTT + 4\cdot DevRTT$$

在平均时长的基础上给超时时长（TimeoutInterval）设定了一定的余量以应对可能的波动偏差，达到较为理想的传输效率。

我们以一段长度为 320 的字符串传输（丢包率：0.1、错误率：0.1）为例，通过以上控制操作动态更新超时间隔，并绘制各中间变量随时间变化曲线如下图 5 所示。

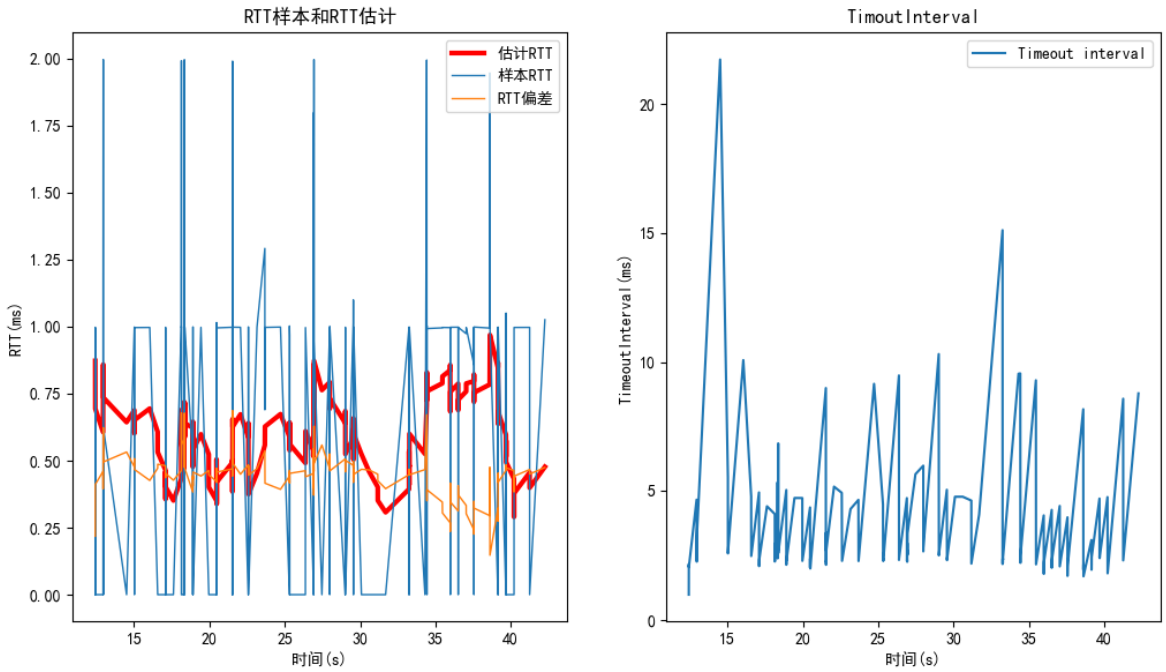


图 5: 超时间隔控制的变量变化

从实验结果可见，SampleRTT 的变化（粗红线）在 EstimatedRTT 的计算（细蓝线）中变得平缓了，而 TimeoutInterval 的值也将随 Estimated 和 DevRTT 的变化而动态更新。

对比有无超时间隔控制的传输过程，分别利用超时间隔设置为固定 1s 和 TCP 动态超时间隔控制，在相同的丢包率和错误率下发送长度为 100 字符串的分组，记录五次发送的总用时，得到结果如下表所示。

表 1: TCP 超时间隔控制与固定超时间隔发送耗时对比（单位：s）

实验次数	1	2	3	4	5	平均值
动态更新超时间隔	22.65	22.97	17.63	12.27	16.71	18.45
固定超时间隔 1s	63.75	55.75	53.85	58.72	56.03	57.62

从上表数据结果也可以直观得出，TCP 协议的超时间隔控制设计从极大程度上避免了静态、固定超时间隔造成的不必要浪费，优化了传输效率。

2.2.2 序列号与控制号机制

与 TCP 一样，TCP 报文段首部包括源端口号和目的端口号，除此以外，TCP 报文段还包括 32 位比特的序号字段、32 位比特的确认号字段等用于实现可靠传输服务的设计。序号建立在传送的字节流之上。因此一**个报文段的序号**是该报文段首字节的字节流编号。TCP 是全双工的，主机 A 向主机 B 发送数据的同时，也许也接收来自主机 B 的数据。**主机 A 填充进报文段的确认号是主机 A 期望从主机 B 收到的下一字节的序号**。用户 A 键入字符发送到远程主机 B，B 将回送该字符的副本 (回显)。A 再发送一个确认收到的报文结束对一个字符的发送过程。通过对 3.5.2 中 Telnet 案例的模拟，实现了 TCP 的序列号与确认号的功能，演示如下:

```
Input sentence:
abcd
Send:  {"Seq": 0, "ACK": 100, "data": "a", "checksum": true}
Receive:  {'Seq': 100, 'ACK': 1, 'data': 'a', 'checksum': True}
Send:  {"Seq": 1, "ACK": 101, "data": "NULL", "checksum": true}

Send:  {"Seq": 1, "ACK": 101, "data": "b", "checksum": true}
Receive:  {'Seq': 101, 'ACK': 2, 'data': 'b', 'checksum': True}
Send:  {"Seq": 2, "ACK": 102, "data": "NULL", "checksum": true}

Send:  {"Seq": 2, "ACK": 102, "data": "c", "checksum": true}
Receive:  {'Seq': 102, 'ACK': 3, 'data': 'c', 'checksum': True}
Send:  {"Seq": 3, "ACK": 103, "data": "NULL", "checksum": true}

Send:  {"Seq": 3, "ACK": 103, "data": "d", "checksum": true}
Receive:  {'Seq': 103, 'ACK': 4, 'data': 'd', 'checksum': True}
Send:  {"Seq": 4, "ACK": 104, "data": "NULL", "checksum": true}

Send:  {"Seq": 4, "ACK": 104, "data": "\n", "checksum": true}
Receive:  {'Seq': 104, 'ACK': 5, 'data': '\n', 'checksum': True}
Send:  {"Seq": 5, "ACK": 105, "data": "NULL", "checksum": true}
```

图 6: 序列号与确认号：Client

```
基于UDP的TCP协议 receiver 端实例化:
{'Seq': 0, 'ACK': 100, 'data': 'a', 'checksum': True}
{'Seq': 1, 'ACK': 101, 'data': 'NULL', 'checksum': True}

{'Seq': 1, 'ACK': 101, 'data': 'b', 'checksum': True}
{'Seq': 2, 'ACK': 102, 'data': 'NULL', 'checksum': True}

{'Seq': 2, 'ACK': 102, 'data': 'c', 'checksum': True}
{'Seq': 3, 'ACK': 103, 'data': 'NULL', 'checksum': True}

{'Seq': 3, 'ACK': 103, 'data': 'd', 'checksum': True}
{'Seq': 4, 'ACK': 104, 'data': 'NULL', 'checksum': True}

{'Seq': 4, 'ACK': 104, 'data': '\n', 'checksum': True}
The received messages through TCP    :
abcd
```

图 7: 序列号与确认号：Server

从模拟程序输出结果可以直观看到，TCP 的序列号与确认号可靠传输机制得到了正确实现。

2.2.3 连接管理机制

TCP 连接管理主要分为两部分：**建立连接 (三次握手)** 与**关闭连接**。

三次握手：

1. 客户端的 TCP 首先先向服务器端的 TCP 发送一个特殊的 TCP 报文段。一个标志位 SYN 被置为 1。
2. 一旦包含 TCP SYN 报文段到达服务器主机，服务器会从该报文中提取出 TCP SYN 报文段，并为该 TCP 连接分配 TCP 缓存和变量，并向该客户 TCP 发送允许连接的报文段。
3. 在收到 SYNACK 报文段后，客户也要给该连接分配缓存和变量。

关闭连接：

客户进程发出关闭指令，客户 TCP 向服务器进程发送一个特殊报文，该报文中标志位 FIN 被置为 1 (在代码中简略为发送 FIN 字符)。服务器接收到该报文后，发送一个确认报文段 (代码中简略为发送 ACK 字符)，并发送自己的中止报文段，其 FIN 也被置为 1 (在代码中简略为发送 FIN 字符)。最后客户对中止字符进行确认。两台主机释放资源。

实现效果 基于已经实现的可靠传输程序，在上一小节已经实现序列号与确认号机制的基础上，通过加入连接建立与关闭机制，我们进一步完善了 TCP 协议的构建与实现。下图展示的，是一个完整的三次握手到关闭连接的过程。

```
基于UDP的TCP协议 sender 端实例化:
Input sentence:
a
SYN sent
client state : SYN_SENT

SYN & ACK received, ACK sent
client state : ESTABLISHED

FIN sent
client state : FIN_WAIT_1

ACK received
client state : FIN_WAIT_2

FIN received, ACK sent
client state : TIME_WAIT

Wait for 30s ...

client state : CLOSED
```

图 8: 连接管理: Client

```
基于UDP的TCP协议 receiver 端实例化:
server state : LISTEN

SYN received, SYN & ACK sent
server state : SYN_RCVD

ACK received
server state : ESTABLISHED

FIN received, ACK sent
server state : CLOSE_WAIT

FIN sent
server state : LAST_ACK

ACK received
server state : CLOSED
```

图 9: 连接管理: Server

从以上代码的运行结果可以看出，server 与 client 端依照 TCP 协议的连接管理机制完成了正确的三次握手并建立可靠连接、最后关闭连接的过程。

2.2.4 拥塞控制机制

拥塞控制是 TCP 协议中十分重要且复杂的议题，由于 IP 层不向端系统提供显式的网络拥塞反馈，因此 TCP 必须使用端到端的拥塞控制而非网络辅助的拥塞控制，我们将抓住 TCP 拥塞控制机制中至关重要的变量（如 LastByteRead、rwnd、ssthresh 等）和过程（慢启动、拥塞避免和快速恢复）对 TCP 拥塞控制机制进行宏观把握，并利用 python 程序进行模拟实现，通过不同初始参数值的设定观察并分析不同情况下 TCP 拥塞控制的运行特点，并作图进行直观呈现。

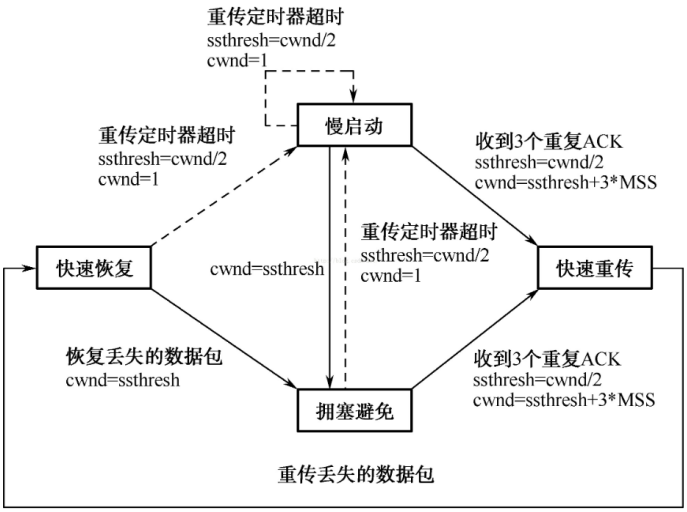
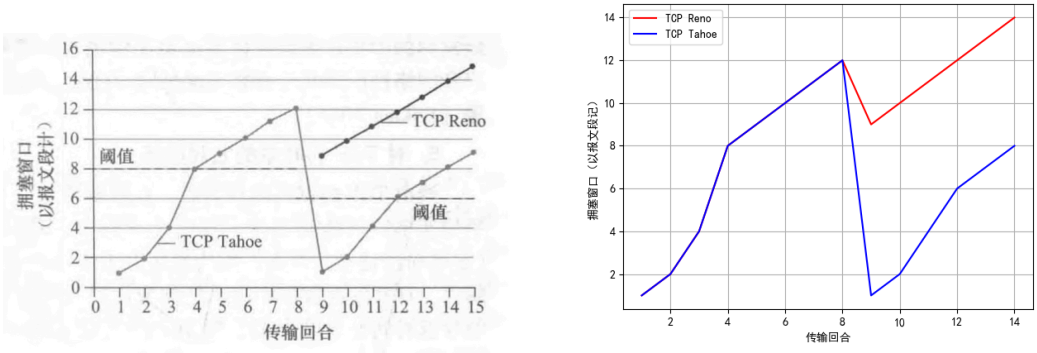


图 10: TCP Reno 拥塞控制的 FSM 描述

通过课程学习我们知道，运行在 TCP 的控制算法主要包括三个部分：慢启动、拥塞避免和快速恢复，整个拥塞控制算法的执行过程可以用以下有限状态机表示。当一条 TCP 连接开始时，cwnd 通常设为较小的初始值（如 1 MSS），在慢启动阶段，cwnd 的值以 1 个 MSS 开始并每当传输的报文段首次被确认就增加一个 MSS，因此，TCP 发送速率起始慢，但在慢启动阶段以指数增长。这种指数增长直到存在一个由超时指示的丢包事件（即拥塞）发生，发送方 TCP 将 cwnd 重新置为 1 并把 ssthresh 的值置为 $\frac{cwnd}{2}$；当下次窗口值达到（超过）ssthresh 后，TCP 将执行拥塞避免模式，即每次增加一个 MSS；最后，当检测到 3 个冗余 ACK 时，TCP 也将结束慢启动并进入快速恢复状态，即阈值减半，TCP Reno 将拥塞窗口设为 $\frac{ssthresh}{2}$。同样地，当超时事件发生或达到阈值，则正常进入慢启动或拥塞避免状态。

通过 python 程序，我们对上述过程进行了模拟，当初始窗口大小为 1，阈值为 8，第八个传输回合出现三个冗余 ACK 超时事件时，下图分别展示了早期 TCP Tahoe 和带有快速恢复（重传）的 TCP Reno 的拥塞窗口演化情况，测试样例与书中展示情形相符。



(a) 教材样例　　(b) 模拟实现

图 11: 样例实现：两种 TCP 拥塞窗口变化

接下来，我们模拟实际传输过程中可能遇到的丢包超时和冗余重传的情形，为 simulator 设置上述两种情形发生的概率为 0.05 和 0.1，在初始窗口大小为 1（MSS），初始阈值（ssthresh）为 25 的条件下传输 250 个回合，得到下图所示的过程图。

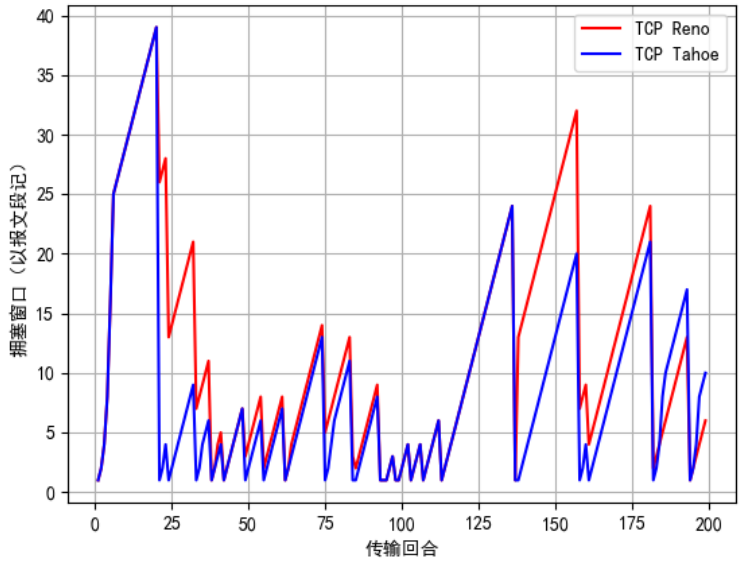


图 12: TCP 实际传输拥塞控制模拟实现

可以看到，在接近实际传输的模拟环境中，TCP 协议采取的拥塞控制算法遵守 **“加性增、乘性减”** 的基本设计原则，与实际拥塞情况匹配良好，以 Reno 为代表的快速重传机制，结合 2.2.1 节探究的超时间隔加倍机制，在避免拥塞的同时兼顾了传输效率的稳定。

拓展学习：TCP 拥塞控制算法发展 1987 年，Van Jacobson 为解决网络拥塞提出 Tahoe；1990 年，快速恢复和快速重传添加到 Reno；1994 年，基于 RTT 判断拥塞的 TCP Vegas 问世；1996 年，SACK 机制在 RFC 2018 中被介绍；1999 年，New Reno 被提出，并引入 SACK 机制；2003 年，TCP Veno；2004 年，FAST TCP，infocom2004；2006 年，Compound TCP，infocom2006；2008 年，TCP CUBIC，采用混合慢启动；2013 年，Sprout，nsdi2013；2013 年，Remy，sigcomm’2013；2015 年，TCP Verus，sigcomm2015；2015 年，PCC，nsdi’2015；2016 年，BBR，ACM queue’2016 ……[13] 我们知道，TCP 协议仅定义框架，也就是发送端和接收端需要遵循的 “规则”。TCP 协议的实现经过多年的改进，有了多个不同的版本。比较重要的有 Tahoe、Reno、NewReno、SACK、Vegas 等，有些已经成为了影响广泛的 RFC 文档，有些则成为了 Unix/Linux 操作系统的标准选项。

作为本次大作业的最后一部分，我们接下来展示了两种经典的其他 TCP 拥塞控制算法，开阔视野，在与我们模拟实现的 Reno 算法对比中进一步理解计算机网络发展创新的多元性。

TCP Vegas 1994 年，Brakmo 提出了一种新的拥塞控制机制 TCP Vegas，从另外的一个角度来进行拥塞控制。从前面可以看到，TCP 的拥塞控制是基于丢包的，一旦出现丢包，于是调整拥

塞窗口，然而由于丢包不一定是由于网络进入了拥塞，但是由于 RTT 值与网络运行情况有比较密切的关系，于是 TCP Vegas 利用 RTT 值的改变来判断网络是否拥塞，从而调整拥塞控制窗口。如果发现 RTT 在增大，Vegas 就认为网络正在发生拥塞，于是开始减小拥塞窗口，如果 RTT 变小，Vegas 认为网络拥塞正在逐步解除，于是再次增加拥塞窗口。由于 Vegas 不是利用丢包来判断网络可用带宽，而是利用 RTT 变化来判断，因而可以更精确的探测网络的可用带宽，从而效率更好。

然而 Vegas 的有一个缺陷，并且可以说致命的，最终影响 TCP Vegas 并没有在互联网上大规模使用。这个问题就是采用 TCP Vegas 的流的带宽竞争力不及未使用 TCP Vegas 的流，这是因为网络中路由器只要缓冲了数据，就会造成 RTT 的变大，如果缓冲区没有溢出的话，并不会发生拥塞，但是由于缓存数据就会导致处理时延，从而 RTT 变大，特别是在带宽比较小的网络上，只要一开始传输数据，RTT 就会急剧增大，这在无线网络中体现的尤为明显。

TCP CUBIC 在 linux 2.6.18 中，默认的拥塞控制算法采用了 CUBIC。我们知道，在 reno 版本的拥塞控制下，进入拥塞避免状态或快速恢复状态后，每经过一个 RTT 才会将窗口大小加 1，那么当链路状况良好时，如果 RTT 很长，reno 达到最佳窗口大小的时间就会有浪费，BIC 实际上属于一种二分搜索拥塞控制算法。其具体算法是：当前链路在网络上因为排队而发生丢包时，链路的当前最佳拥塞窗口肯定是小于丢包时的拥塞窗口的，我们称丢包时的拥塞窗口大小为 W_{max}，同样，BIC 也采用乘法减小的方式减小窗口，我们称这个因子为，同时，我们称减小后的窗口为 W_{min}，则有 $W_{min} = *W_{max}$，我们认为乘法减小后的窗口应该是小于最佳拥塞窗口的，因此，对于链路当前最佳拥塞窗口 W 来说，我们有 $W_{min} < W < W_{max}$，reno 采用加法去搜索 W 的方式较慢，因此 BIC 采用了二分搜索的方式去找到这个 W。[14]

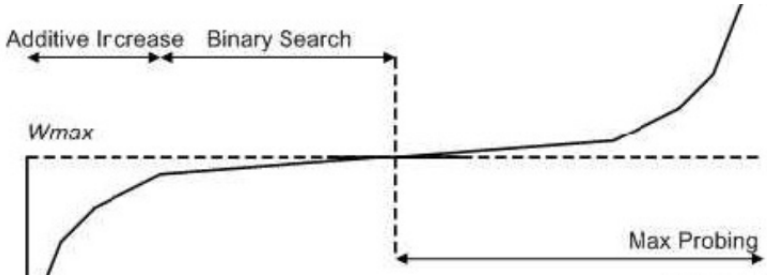


图 13: BIC 二分增窗示意图

在 BIC 的基础上，CUBIC 既希望保证高效性，又希望维护公平性。考虑任意一个奇数阶的多项式图像均可以拟合 BIC 的图像，对于公平性而言，既然 reno 和 BIC 都以 RTT 为单位争抢资源，把窗口的调整过程从以 RTT 为单位的思路出来，进而通过调整时间的步长来维持公平性。CUBIC 通过这个公式给出了解决方案：

$$W(t) = C(t-K)^3 + W_{max}$$

至此，我们粗略地定性分析了这两种传统算法之外的 TCP 拥塞控制算法，看到了它们在不同场景和应用需求下各具特色的实现效果和方案，这也更进一步加深了我们对计算机网络中各类协议、算法日新月异的发展进程的理解。

3 总结

通过一学期的学习，我们对计算机网络这门课程所涵盖的内容有了初步掌握，也对计算机网络这一广阔的研究、应用领域有了基本的认识和基础知识储备。结合本篇结课论文，我们在前半（综述）部分按计算机网络传统分层结构依次对各层核心内容、重点知识进行了梳理、思考与整合，并对每一层所涉及的知识内容，查阅相关文献，学习并努力理解相关前沿技术进展的细节，在文中相应段落进行了呈现。

接着，在应用实践（大作业）部分，我们对传输层的相关内容进行了较为深入、透彻、完整的研习和实践。从基于 RDT 和流水线传输的 GBN、SR 协议实现与分析，到对网络世界至关重要的 TCP 协议各大基本特征的理解与实现，我们基本完成了对 TCP 所涉及的包括连接管理、超时间隔、确认号机制和拥塞控制等几大板块的实现、模拟和评估。

最后，我们还基于 TCP 的发展历程，简要探索了以拥塞控制算法为代表的网络协议、算法分支和前沿，更进一步理解了计算机网络中复杂精妙的平衡与妥协的设计智慧与技术高峰。

感谢老师和助教一学期以来给予的指导、帮助和关心！

参考文献

[1] “Application Layer | Layer 7”. The OSI-Model. Retrieved November 5, 2019.

[2] https://en.m.wikipedia.org/wiki/Application_layer

[3] James F.Kurose, Keith W.Ross, Computer Networking: A Top-Down Approach, 7th edition, 2018

[4] 戚琦，申润业，王敬宇.GAD：基于拓扑感知的时间序列异常检测 [J]. 通信学报，2020，41(6)：152-160.

[5] 刘锦康，赵征. 基于拓扑感知的流媒体编码和应用层组播系统设计 [J]. 现代电子技术,2021,44(16):1-6.DOI:10.16652/j.issn.1004-373x.2021.16.001.

[6] 许自程，冯陈伟. 基于树莓派的远程网络视频监控系统 [J]. 电视技术，2018，42(10)：92-97.

[7] ZHANG B，LIU Z，DONG S. IAP-Based Self-Learning Real-Time Application Layer DDoS Detection Method on Storm Platform[C]// 2019 IEEE International Conference on Parallel and Distributed Processing with Applications，Big Data and Cloud Computing，Sustainable Computing and Communications，Social Computing and Networking. Piscataway：IEEE，2019：912-919.

[8] 李颖之, 李曼, 董平, 周华春. 基于集成学习的多类型应用层 DDoS 攻击检测方法 [J/OL]. 计算机应用:1-9[2022-05-22].http://kns.cnki.net/kcms/detail/51.1307.TP.20220416.0837.004.html

[9] MITOLA J MAGUIRE G Q. Cognitive Radio：Making Software Radios More Personal[J]. IEEE Wireless Communications-1999-6(4)：13—18.

[10] 李志高, 周音, 孙学斌, 周正. 认知无线电网络中一种改进的传输层协议 [J]. 无线电工程,2011,41(10):1-3+16.

[11] 李敏. 量子保密通信网络路由算法研究 [D]. 西安: 西安电子科技大学, 2017.

[12] 徐雅斌, 张梅舒, 李艳平. 基于回溯的 QKD 网络随机路由选择算法研究 [J]. 电子科技大学学报,2021,50(04):565-571.

[13] 深入理解 TCP 拥塞控制——从 BIC 到 CUBIC，2021-04，https://zhuanlan.zhihu.com/p/366392032

[14] TCP 拥塞控制算法的演进：2015-10-03, https://blog.csdn.net/xuyongshi02/article/details/48867633