
华中科技大学计算机学院

《计算机通信与网络》实验报告

姓名 胡思勖 班级 计卓 1501 学号 U201514898

项目	Socket 编程 (40%)	NS2 实验 (20%)	CPT 实验 (20%)	平时成绩 (20%)	总分
得分					

教师评语：

教师签名：

给分日期：

目 录

实验一 SOCKET 编程实验	1
1. 1 环境.....	1
1. 2 系统功能需求	1
1. 3 系统设计	1
1. 4 系统实现	15
1. 5 系统测试及结果说明.....	38
1. 6 其它需要说明的问题.....	45
实验二 基于 NS2 的协议分析实验.....	49
2. 1 环境.....	49
2. 2 实验要求	49
2. 3 实验步骤说明及结果分析.....	50
2. 4 其它需要说明的问题.....	74
实验三 基于 CPT 的组网实验	75
3. 1 环境.....	75
3. 2 实验要求	75
3. 3 基本部分实验步骤说明及结果分析.....	77
3. 4 综合部分实验设计、实验步骤及结果分析	86
3. 5 其它需要说明的问题.....	93
心得体会与建议	94
4. 1 心得体会	94
4. 2 建议	94

实验一 Socket 编程实验

1.1 环境

1.1.1 开发平台

处理器: Intel® Core™ i7-6700HQ CPU @ 2.60GHz 2.59GHz
内存大小: 8.00 GB (7.89GB 可用)
操作系统: Windows 10 Pro 64 位 10.0.15063
开发平台: Msys2 Mingw64
集成开发环境: Qt Creator 4.4.1
第三方组件: Qt 5.9.2 (x86_64-little_endian-llp64 static)
 AES lib (implemented by Chris Lomont) version 1.0
编译器: Mingw64 gcc 7.2.0 (Rev1, Built by MSYS2 project)
链接器: Mingw64 ld 2.29.1
调试器: Mingw64 gdb 8.0.1
Qt 预处理器: Mingw64 moc 5.9.2
Qt Make 工具: Mingw64 qmake 3.1 (static version)
发布工具: Mingw64 windeployqt 5.9.2

1.1.2 运行平台

处理器: Intel® Core™ i5-4570 CPU @ 3.20GHz 3.20GHz
内存大小: 8.00 GB (7.89GB 可用)
操作系统: Windows 7 (7.0) 64 位 6.1.7601 Service Pack 1
第三方组件: 无 (全静态链接无需第三方库)

1.2 系统功能需求

工具包括服务器端和客户端。
具备用户注册、登录、找回密码功能（基于 TCP 协议）。
两个用户如果同时在线，采用点到点通信方式进行聊天，信息不需要通过
服务器中转，服务器也不保存（基于 TCP 协议）。

-
- 支持离线消息（基于 TCP 协议）。
 - 支持点到点可靠文件传输（基于 UDP 协议）。
 - 存储在服务器端的数据需要进行强加密。
 - 支持不少于两组用户同时在线交流和传输文件。
 - 文件传输具有良好的性能，能够充分利用网路带宽。
 - 人机交互友好，软件易用性强。

1.3 系统设计

1.3.1 总体架构

从服务端和客户端的功能分析，服务端需要具有同时与多个客户端通信的能力，客户端需要同时具有与服务端通信的能力和与多个其他客户端通信的能力（P2P 通信），也就是说二者都需要具有与多个节点通信的能力。此外，根据系统需求，服务器的负载能力不需要很大，能够同时与上百个客户端通信就已足够，而在用户使用客户端的过程中极少有可能同时与百位数的客户端通信，因此在这种情况下客户端和端所需要的负载能力不会相差很多。

在这两个前提下，本着不将同样的功能重复实现两遍的原则，在设计中将客户端和服务端的通信模块抽象出来，作为一个单独的通信模块，客户端和服务端使用同一通信模块，总体框架如图 1.1 所示，图中处于下方的模块为处于上方的模块提供服务。这样不仅有利于模块的管理和修改，还有利于客户端和服务端间应用层间通信协议的统一。

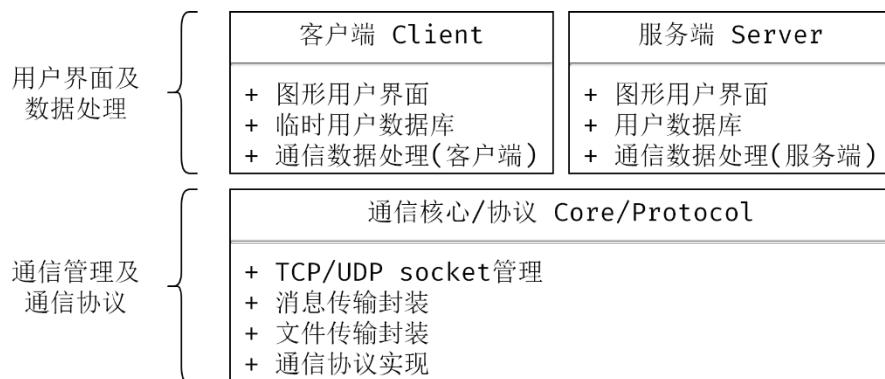


图 1.1 服务端与客户端总体结构

总体上而言，整个项目总体上分为三个模块：核心模块，客户端模块和服务端模块，每个模块为一个独立的工程，可单编译调试。核心模块提供 TCP/UDP socket 管理、对于消息传输的封装（几乎 TCP）、对于文件传输的封装（基于 UDP）以及应用层通信协议的实现，这一核心模块将被编译为静态库供客户端与服务端使用。客户端实现了与用户进行交互的图形界面，临时用户数据库（用

于存储在线用户信息), 以及与客户端有关的通信数据处理的功能。服务端同样实现图形界面, 用户数据库以及通信数据处理的功能。服务端与客户端中的用户数据库与通信数据处理实现的具体功能不同, 与客户端/服务端的角色有关, 因此不宜像通信核心那样抽象并实现为一个独立的、为客户端与服务端所共享的模块。

1.3.2 通信核心/协议部分架构

核心/协议模块包括连接器 BaseConnector、文件传输器 BaseFiletransceiver、与动作描述符 OperationDescriptor 三个模块构成。连接器用于建立并管理 TCP 连接, 文件传输器用于发送或接受文件, 而动作描述符用于以信号的方式提示调用者动作的执行情况。在这里动作是指核心部分的调用者所进行的每一次方法调用, 每当进行一次方法调用时, 会要求调用者传入一个全局唯一的动作描述符, 核心模块则会通过与描述符的通信来向外部模块发送信号。架构如图 1.2 所示, 图中的三个子模块只给出了核心模块暴露给外部模块的接口部分而没有写出内部接口以及其他成员。

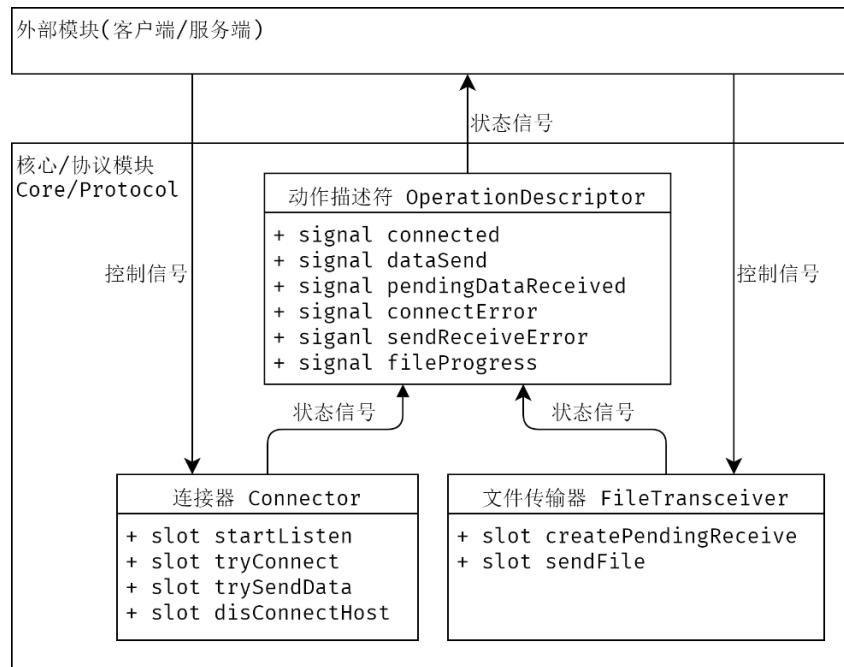


图 1.2 核心/协议模块结构

由于通信模块与客户端或服务端模块处于两个不同的线程上, 因此不能直接通过跨线程调用函数的方式通信, 否则会有线程安全问题, 在核心模块中模块内与对于外部模块的通信全部通过信号来实现。连接器通过接收控制信号对外提供 TCP 监听、TCP 连接、发送消息以及断开 TCP 连接的服务; 文件传输器通过接收控制信号对外提供创建文件发送请求和发送文件的服务。连接器和文件传输器的接口要求外部模块每进行一次请求需要传入一个全局唯一的动

作描述符，当连接器和文件传输器处理完某一项请求或有需要外部注意的信息时，会把这些信息发送给动作描述符，由动作描述符实例进行分拣和转换后再发送给外部模块。

在这里，使用动作描述符而不是直接从连接器和文件传输器向外部发送信号，首先是出于对信号与操作对应的考虑。由于在核心模块中使用的 QTcpSocket 和 QUdpSocket 都是异步的，而 Qt 对于 socket 发出的信号的封装中并没有指出这一信号是由哪一个操作发出的，也就是说，若对于 socket 进行了两次写操作，然后收到了一次错误信号，那么并不知道这一信号是哪一个操作发出的。如果直接将这一信号转发给外部模块将给外部模块造成一定的困难，因此需要使用一个动作描述符对这些信号进行分拣后再转发给外部模块。其二是出于对信号的翻译与限制的考虑。由 Qt 所封装的 socket 发送的信号种类很广泛，而有些信号并不需要被外部模块获取或处理，有些信号又可以合并为一个信号再转发以减少调用者的负担，而使用动作描述符可以轻易地达成这一点。最后是出于对可扩展性和易用性的考虑。若需要在本核心模块中添加新的模块，其信号的处理模式可以直接参照已经实现的连接器和文件传输器，而外部模块对于状态信号的处理依然可以通过动作描述符进行，同时减少了开发者和模块使用者的负担。

连接器的主要作用是管理 TCP Socket，为其创建连接，保证连接正常，断

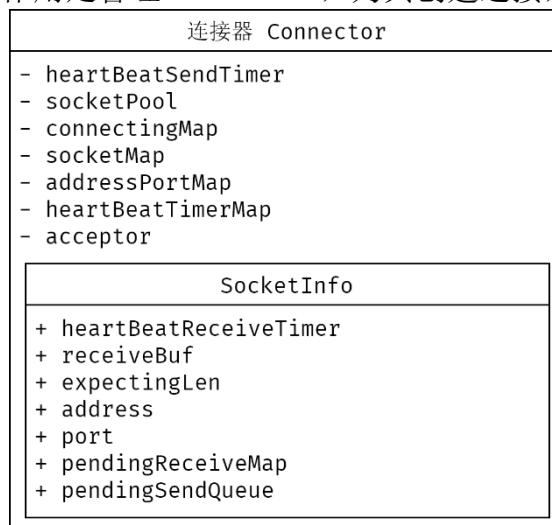


图 1.3 连接器内部结构

开连接以及利用 socket 发送消息。为了实现管理的功能，除了图 1.2 给出的对外部模块暴露的接口之外，其内部还包含了一个 SocketInfo 类，以及多个 SocketInfo 类型的 map，如图 1.3 所示（图中仅给出其成员变量，未给出类方法）。图中，SocketInfo 是用于存储已经连接的 socket 的信息的结构体，其中，heartBeatReceiveTimer 用于接收心跳包的定时器，receiveBuf 是用于接受消息的缓冲区，expectingLen 为当前所接收的消息的长度，address 和 port 为此 socket

所连接的 ip 地址和端口，而 pendingReceiveMap 和 pendingSendMap 将要接收的消息和将要发送的消息的缓冲区。

连接器内 socketPool、connectingMap、socketMap、addressPortMap、heartBeatTimerMap 均为 SocketInfo 类型的容器，用于存储和管理 socket。socket 实例都存储于 socketPool 中，其他容器都存储指向 socket 实例的指针。connectingMap 用于存储正在进行链接的 socket，当 socket 连接完成后会将其移动到 socketMap、addressPortMap 以及 heartBeatTimerMap 中，而这三个关联型容器分别用于以 socket 指针、socket 所连接的地址和端口以及 socketInfo 中的心跳包定时器为键来快速寻找 socketInfo 值。使用三个 map 而不是一个是为了减少根据不同的键查询值的时间。此外，连接器内还包括一个 heartBeatTimer 和一个 acceptor，heartBeatTimer 用于定时向所有已经连接的 socket 中写入心跳包，而 acceptor 则是用于监听的端口，用于接受来自其他终端的连接（在服务端上用于接受来自客户端的连接，在客户端上用于接受 P2P 连接）

文件传输器的结构与连接器的结构较为相似，不同之处在于文件传输器管理的是简要发送和接收的文件而不是 socket，其结构如图 1.4 所示（图中仅给出其成员变量，未给出类方法）。类似于连接器中的 SocketInfo 类，文件传输器

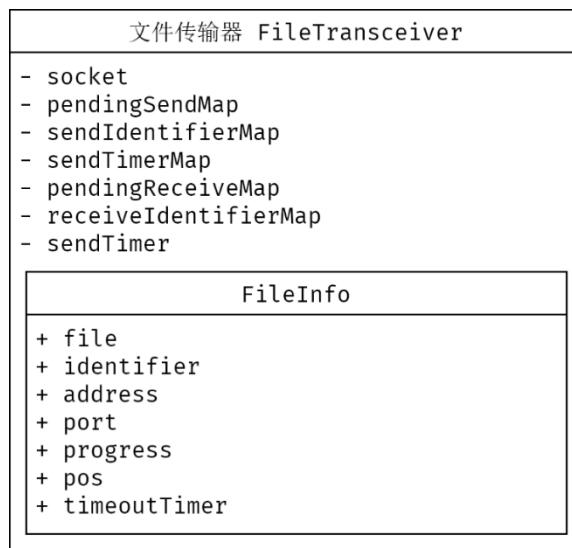


图 1.4 文件传输器内部结构

内包含一个 FileInfo 类，用于存储文件信息。其中，file 为 Qt 文件类的实例，identifier 为一个全局唯一的 64 位整型变量用于区分每个文件，特别是当有两个同名的文件被传输时需要使用 identifier 来区分，address/port 为对等端 socket 的 ip 地址和端口，progress 为一个 bitArray，用于标记文件哪些部分已经被传输完成，progress 则为 progress 中 1 的个数与其总长度的比值，即文件传输的进度。timeoutTimer 为超时计时器，当此计时器超时表明文件传输超时，放弃本次传输。

在文件传输器内，socket 为用于收发文件的端口，所有收发的数据包均通过这个 socket 读写。pendingSendMap、sendIdentifierMap、sendTimerMap 为一组 FileInfo 类型的容器，其中 pendingSendMap 存储的是将要或正在被发送的文件所在的 FileInfo 类的实例，另外两个 map 存放的均为指向实例的指针，用于快速通过 FileInfo 中的 identifier 以及 timer 索引到 FileInfo。pendingReceiveMap 和 receiveIdentifierMap 为另一组 FileInfo 类型的容器，其功能与前一组容器的功能一致，但其存储并管理的是接收的文件。此外，本类还包含一个 sendTimer 定时器，用于在发送文件时定时向 socket 中写入文件数据。调整其超时间隔即可以影响文件的发送速率以及丢包率。

对于动作描述符 OperationDescriptor 而言，其结构较为简单，成员变量仅有 id，其为一全局唯一的 64 位无符号整型，用于区分不同的描述符。其余的类方法大部分是将从连接器和文件传输器接受到的信号映射到发送给外界的信号。

1.3.3 客户端架构

客户端主要由三个部分组成：图形用户界面 GUI，客户端核心 ClientCore 和客户端数据库 DataBase 组成。图形界面包括登陆界面、密码修改界面、注册

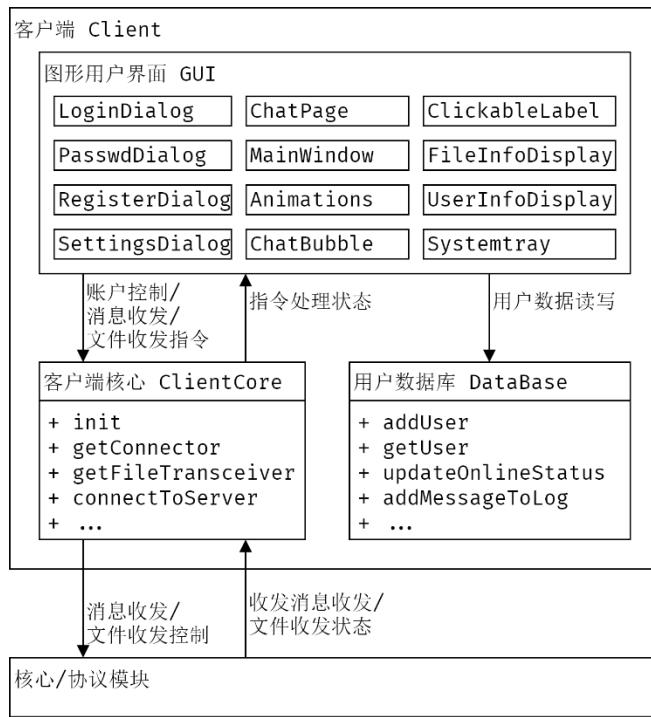


图 1.5 客户端架构

界面、设置界面、聊天主界面、动画库、文件信息显示界面、系统图标库、标题栏库以及用户信息显示界面等多个子模块构成，其主要功能为与用户进行交互并将交互数据发送给用户核心进行处理。客户端架构如图 1.5 所示。图形用

户界面在不同情况下采取不同的消息处理机制：当数据和指令能够直接被核心模块处理时，图形界面模块首先调用 `ClientCore::getConnector` 或 `ClientCore::getFileTransceiver` 方法获得连接器或文件传输器的单例实例后直接将消息发送给文件传输器，否则调用 `ClientCore` 对应的方法将数据交给用户核心处理。这些不能直接被处理的指令为客户端特有指令，包括连接到服务器，设置用户账户，获取服务器地址以及端口等。

用户核心采用单例模式实现，对于连接器或文件传输器所返回的数据处理结果，用户核心进行二次处理，处理为图形用户界面可以理解的数据格式后以信号的方式发送给图形用户界面，然后反映在显示屏上以实现与用户的交互。

客户端的数据库主要用于存储用户列表及用户的信息，包括用户邮箱、用户名、在线状态、用户发送的消息列表、文件列表。若用户在线，还会存储此在线用户的监听用 `socket` 的 ip 地址和端口。由于客户端的用户列表是在用户登陆时从服务端取得的，因此数据库没有继承序列化和持久化的功能。客户端用户数据库如图 1.6 所示（只标明成员变量以及内嵌类），其内部包含一个用户信息类 `UserInfo`，用于存储上述用户信息，而 `emailMap` 和 `addrPortMap` 则分别用于快速通过用户邮箱和用户的监听 ip 地址及端口快速索引到用户信息，这样可以在用户数量较大的时候依然保持较高的索引性能。此外，数据库提供了一系列的方法来操作数据库，包括向数据库中添加用户，修改用户在线状态以及其他信息，从数据库中删除用户等功能。



图 1.6 用户数据库内部架构

1.3.4 服务端架构

服务端架构与客户端类似，不同之处在于：在客户端需要进行 P2P 连接与其他客户端进行通信时，需要主动连接对方的监听端口，而服务端只被动接受来自客户端的连接并利用已经连接的 `socket` 与客户端进行通信，而不会主动连接客户端的监听 `socket`。服务端的内部架构如图 1.7 所示，与客户端类似，服务端内部也含有三个部分：图形用户界面 GUI，服务端核心 `ServerCore` 和服务

端用户数据库 DataBase。不同的是，在服务端核心与用户数据库之间还有一个数据处理器模块 DataProcessor，用于处理，用于处理来自服务端核心的数据、操作用户数据库。此外，用户界面的功能也被局限为仅用于显示数据与修改数据库密码，其主要功能为显示数据库中的数据及变化而不是与用户的交互，因此不像客户端那样由核心发送信号来影响图形用户界面的变化，而是直接由用户数据库发送信号，每当数据库发生改变时发送信号给图形用户界面以更新其内容。

对于服务端核心而言，其功能较为简单，仅用于初始化整个服务端（包括初始化通信核心模块，开始监听端口），以及将通信核心发送过来的数据和信号

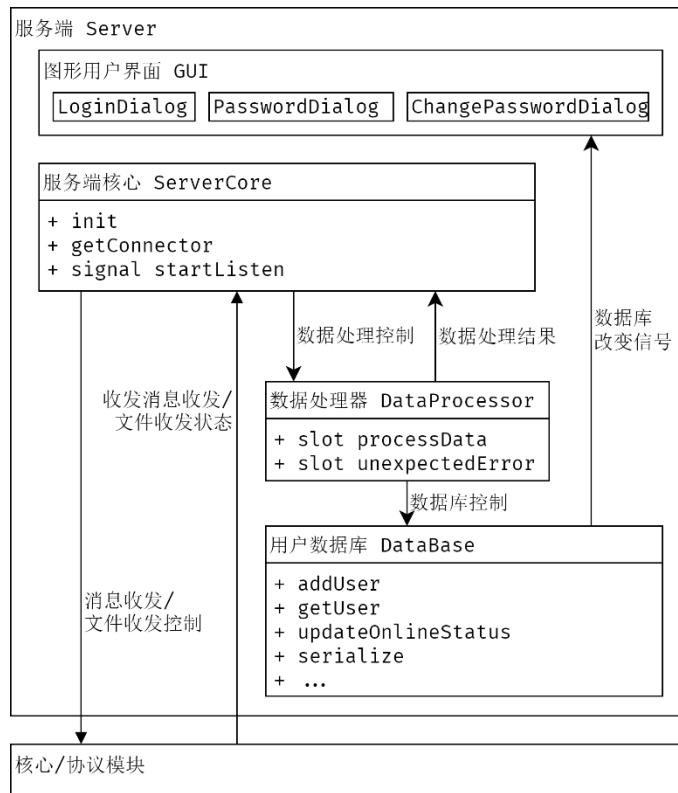


图 1.7 服务端内部架构

经过初步处理后转发给数据处理器。数据处理器接受来自于服务端核心的数据和信号并根据协议内容以及服务端需求进行处理，并在必要的时候对数据库进行操作，然后将数据处理结果返回给服务端核心，由服务端核心将处理结果发送给客户端。

服务端的用户数据库与图 1.6 所示的客户端用户数据库结构几乎一致，但其内嵌用户信息 UserInfo 类中包含了更多的内容，包括 hash 过的密码 passwd、密码找回用的安全问题 safeQuestion、hash 过的答案 safeQuestionAnswer、连接端口 connectPort、服务器暂存消息列表 cachedMessage。值得注意的是连接端口 connectPort 与监听端口 port 是不一样的。由于服务器只与主动连接成功的

客户端通信，因此直接向客户端连接服务器的端口向服务器写数据，而不需要主动连接客户端的监听端口。`connectPort` 表示的就是客户端主动连接服务器的连接端口，用于服务器向客户端发送消息。此外，相较于客户端的用户数据库，服务端的用户数据有序列化与持久化的需求，因此加入了序列化方法与 AES 加密方法，用于将数据库内容加密后存储于磁盘上，在服务器重新启动时再从磁盘中读取数据库内容。当数据发生变化时，会发送信号告知用户界面数据库发生变化，并以增量形式表示变化数据，从而将数据库变化反映在图形界面上。

1.3.5 协议设计

TCP 消息传输协议设计

在应用层 TCP 消息传输协议的设计中，采用变长报文的形式，以适应不同情况下不同的报文长度。数据报文格式如图 1.8 所示。由 19 字节的首部和变长的数据组成。首部中，依次为 8 字节的请求编号 `ReqNum`、3 字节的报文类型 `Header`、8 字节的数据长度 `DataLen`。请求编号用于区分不同的请求，由请求发出方保证对于自身而言每个请求是全局唯一的，即对于请求发出方而言没有两个等待处理的请求的编号是相同的。区分请求的目的在于简化请求与请求应答的对应关系。虽然 TCP 保证了请求应答报文不会乱序到达或部分缺失，但不能保证报文一定能够到达。即使依赖于 TCP 的可靠传输也肯能导致程序逻辑混乱。在连接断开时，请求发出方会尝试重新连接，连接重新建立后请求应答方可能会将原来请求的处理结果再次发送，在这一重新建立连接的过程中可能导致请求处理结果被请求应答方丢失，导致某些请求未被应答，从而导致对应关系的混乱。

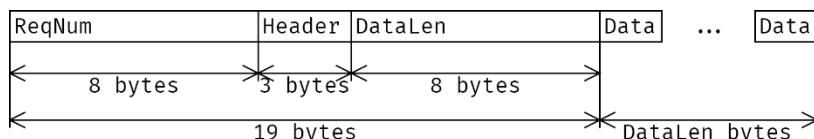


图 1.8 报文数据格式

这种在处理请求过程中连接断开的一个例子如图 1.9 所示，若两个同类型但内容不同的请求先后发出（如注册用户请求，两次使用的密码不同），在请求被成功接收到后连接断开，此时请求 1 处理完成后返回，由于连接断开发送失败，服务器不能保证客户端一定会重新连接，因此不能直接在内存中缓存结果，否则可能导致资源泄漏，于是将结果丢弃。在连接再次重新建立后返回请求 2 的处理结果。此时对于请求发送方而言，它并不知道请求 1 的处理结果返回失败，因此它不知道重新连接后返回的是哪一个请求的结果，将导致处理困难，这在请求数量较多时会带来更多处理困难。而使用请求编号则可以较为简单地解决这个问题：对于每个请求进行唯一的编号，请求应答方处理完请求后使用

相同的编号将处理结果返回即可。

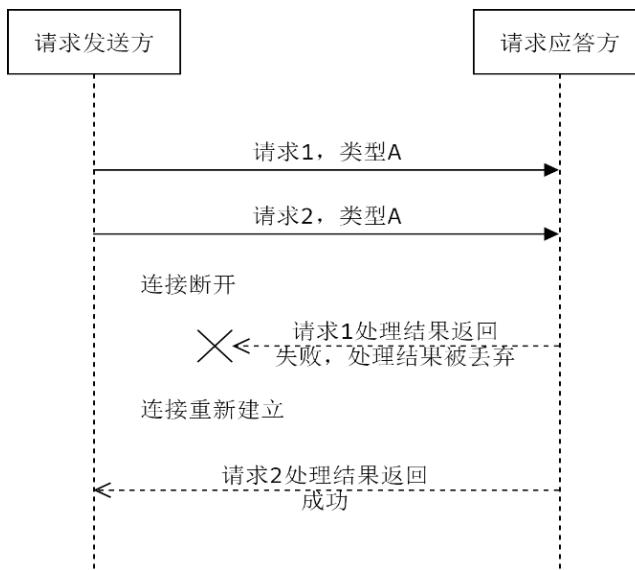


图 1.9 请求过程中连接断开示例

报文首部中的三字节报文类型 Header 用于区分不同的报文类型，其可取的值如表 1-1 所示。取值中的第一个字母限定了报文的类型，R 代表注册(Register)，L 代表登陆(Login)，P 代表密码找回(Password-findback)，U 代表用户状态(User-status)，M 代表消息(Message)，F 代表文件(File)，H 代表心跳包(Heart-beat)。第二个字母则限定了报文的子类型，其中 R 代表请求(Request)，A 代表应答(Answer)，C 代表客户端间消息(Client-to-client)，B 代表广播消息(Broadcast)，T 代表转发(re-Transmit)。取值中的第三个字符代表了请求的阶段。对于部分需要通过多个请求→应答过程完成的事件，需要用 Header 中的第三个字符标明请求的阶段，而对于可以一次完成的请求，这一字符始终为 1。

表 1-1 应用层报文类型取值

目录	请求类型	客户端→服务端	服务端→客户端	客户端→客户端
账户控制	注册	RR1	RA1	
	登陆	LR1	LA1	
	密码找回	PR1/PR2/PR3	PA1/PA2/PA3	
用户状态变更		UB1		
消息收发	离线消息	MR1	MA1	
	在线消息		MT1	MC1
文件传输	离线文件	FR1/FR2	FA1/FA2	
	在线文件			FC1/FC2
其他	心跳报文	HB1	HB1	HB1

数据部分采用 Json 格式传输，Json 格式应用广泛，Qt 中也集成了 Json 解析库，便于程序的编写，同时有利于程序的扩展。不将报文类型 Header 置入 Json 文本中是为了加快对于数据的处理速度，对于每个报文可以减少一次从 Json 中取出元素的时间，并且对于错误的、不存在的或恶意篡改的报文类型可以直接识别并丢弃，减少了对 Json 的解析。此外，心跳包发送较为频繁，若对每个心跳包进行 Json 解析则会增大客户端和服务端的负担，而采用 3 字节的 Header 只需比对第一个字符即可，极大的减少了其处理时间。对于不同的类型的报文，其数据部分所必须包含的 Json 需要包含的键值对如表 1-2 所示，其中离线文件由于时间不足暂未实现，其他报文格式及处理均已实现。Json 文本中包含的表中所标明的键值对之外的键值对将会被忽略。

表 1-2 应用层报文数据部分定义

Header/报文类型	Json 数据键值对定义		
	键	值类型	描述
RR1/注册请求	email:	string	邮箱
	username:	string	昵称
	passwd:	string	密码（已 hash）
	question:	string	安全问题
	answer:	string	安全问题答案（已 hash）
LR1/登录请求	email:	string	邮箱
	passwd:	string	密码（已 hash）
	port:	int	监听端口
PR1/更改密码请求 1	email:	string	邮箱
PR2/更改密码请求 2	email:	string	邮箱
	answer:	string	安全问题答案（已 hash）
PR3/更改密码请求 3	email:	string	邮箱
	validate:	int	超时验证随机数
	passwd:	string	新密码（已 hash）
MR1/离线数据请求	email:	string	发送者的邮箱
	targetEmail:	string	目的用户的邮箱
	message:	string	需要缓存的离线数据
RA1/注册应答	reply:	string	注册成功/失败原因
	reply:	string	登录成功/失败原因
	friends: {	Json	好友列表
	<email>: {	Json	好友信息（键为好友邮箱）
	username	string	此好友昵称
	isonline	bool	此好友是否在线
	address	string	好友的 ip 地址（如果在线）
	port	int	好友的监听端口（如果在线）
	}		// 好友信息结束
	...		// 其他好友信息
	}		// 好友列表结束

	message: { <email>: [<data> string ...] ... }	Json	离线消息列表 好友消息（键为好友邮箱） 消息内容 // 更多消息内容 // 好友消息结束 //更多好友消息 // 离线消息列表结束
PA1/更改密码应答 1	reply: string safeQuestoin: string		成功/失败原因 安全问题（如果处理成功）
PA2/更改密码应答 2	reply: string validate: int		成功/失败原因 超时验证随机数
PA3/更改密码应答 3	reply: string		密码更改成功/失败原因
MA1/离线数据应答	reply: email: string isonline: bool address: string port: string		离线数据缓存成功/失败原因 状态变更的用户 新的在线状态 新的 ip 地址（如果在线） 新的监听端口（如果在线）
UB1/用户状态广播	email: string message: string		消息发送者的邮箱 消息内容
MC1/P2P 消息	filename: string filesize: int port: int		文件名 文件大小 发送方的文件发送端口
FC1/文件发送请求	reply: string identifier: int port: int		同意接收/失败原因 接收描述符（发送文件时使用） 接收方的文件接收端口
FC2/文件发送应答			

对于所有的请求消息（Header 第二字节为 R 的消息），都有一个对应的应答消息。请求接收方会尽量保证消息正确处理并且应答成功发送。若应答未成功发送，则请求接收方需要回滚到消息处理之前的状态，以避免服务器和客户端状态不一致。应用层报文中，文件发送报文只是为发送文件做了部分准备，包括互相告知对方自己的文件收发端口，发送方告知接收方文件名以及文件大小，接收方告知发送方是否接收等，并没有使用 TCP 发送文件。

在应用层协议的设计中，假设所有 TCP 连接都采用长连接的形式，对于请求应答方相同的多个连续请求均在同一个连接中完成，且除非请求发送方主动断开连接，否则双方将通过定时发送心跳包的方式使连接一直保持连接畅通。在网络层，数据报可能从任何地方被切分成多个部分并分别包装到不同的 TCP 包中发送，发送方发送一个报文时，接收方可能需要接收多次。此时接收方将采取如下的策略接收报文：首先尝试从 socket 中读取 19 字节的首部，如果已经读取了 m 字节但 m<19 则继续尝试读取 19-m 长度的报文，直到接收到整个首部。如果首部接收完成，则从首部中读取报文长度 DataLen，然后继续尝试从 socket 中读取 DataLen 长度的报文，直到报文读取完成，进入下一个循环，

重新开始等待接收首部。

UDP 文件传输协议设计

在两个用户通过消息传输协议做好文件发送准备后，文件发送方会得知接收方是否同意接受文件，若同意，发送方会得知的文件接收端口以及文件接收描述符，而接收方会知道发送方简要发送的文件的文件名以及文件大小。文件接收描述符用于处理接收方同时接收多个同名文件的情况：由于所有文件片段均通过一个 UDPsocket 读取，若使用文件名区分不同的文件不仅可能导致无意义的增加文件报文长度，更会导致接收同名文件时的冲突，而使用文件描述符，由文件接收方保证每个描述符全局唯一并将文件描述符映射到各个接收的文件则可以解决这一问题，发送方在发送文件片段时只需将文件描述符至于各文件报文的首部即可。

文件报文的结构如图 1.10 所示其中。在首部中，最开始 4 字节的为校验和 crc32，将其至于首位是便于计算与检验整个报文的 crc32 值，不需将报文重新组合。接下来的 1 字节为控制字节 control，用于标识报文的类型，包括文件数据段报文('D', Data)，数据段接受成功报文('A', Acknowledge)，文件接收完成指示('E', All-received)，以及停止连接指示('F', Finish)。接下来的 2 字节的 identifier 即为上述用于区分文件的文件描述符，接下来的 4 字节 seqNum 标识文件片段编号。上述 11 字节构成了报文的首部，接下来的为 segmentSize 大小的定长数据。segmentSize 为一不小于 256 字节的常量值，其上限与平台相关。这样构造的报文至少支持大小为 $2^{32} \times 256\text{Bytes} = 1\text{TBytes}$ 大小的文件传输，完全能够满足日常文件传输需求。文件将按照 segmentSize 大小进行分段，然后将每段包裹进一个独立的报文中进行传输。文件大小可能不是 segmentSize 的整数倍，因此最后一段可能不能填满 segmentSize 大小的字段，此时多余的字段用 0 填充。由于在文件传输之前已经知道了文件的总大小，因此不会将最后一段多填入的 0 写入文件中。

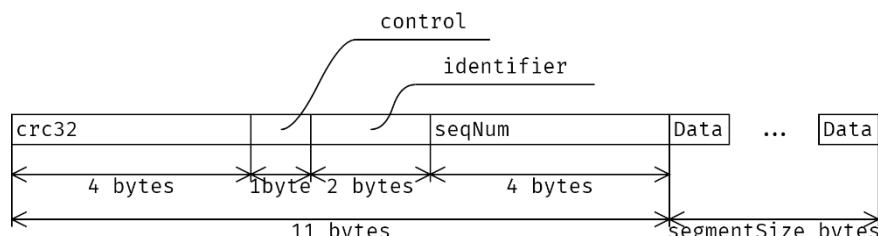


图 1.10 文件报文数据格式

由于需要尽量利用带宽以达到最高文件传输速率，采用如下的传输策略：发送端以一固定速率发送其认为接收端还未收到的片段，接收方每接收到一个片段，向发送方发送一个相应片段的 ACK (ACK 的 seqNum 字段即为其接收到的片段的编号)，发送端使用一个 bitArray 来存储哪些片段已经确认收到，对

于收到 ACK 的片段，在 bitArray 中将对应的编号置为 1，对于发送方已经确认收到的片段不再发送。

在文件传输结束时，收发双方需要对文件的传输状态具有一致的认可，即发送方需要知道文件已经完全被接收方接收才能够停止文件的传输，而接收方需要确认文件完全接收到才能够停止接收文件并释放资源。要做到这一点，只要发送方接收到所有片段的 ACK 就停止发送即可，这样就已经保证了接收方已经接收到所有的片段。

但使用这一策略有可能出现收发双方对于文件接收状态的认知不一致的情况。比如接收方接收到了最后一个片段并发送了这个片段的 ACK，然后释放此文件的相关资源，但这一 ACK 有可能丢失，导致发送方认为文件没有被完全接收，而继续不断发送没有收到 ACK 的片段，导资源泄漏。这当然可以使用超时解决，但仍旧会导致接收方已经接收到完整的文件但发送方认为接收方未完整接收文件（超时导致的发送失败）。这实际上是一个两军问题¹，对于这一问题只能降低不一致的概率而无法达成完全的一致。TCP 中的三次握手四次挥手也同样只能提高双方状态一致的概率。因此本协议在文件传输完成时增加了 2 次确认：当接收方接收完所有的文件片段后向发送方发送一个 All-received 报文，接收方在接收到 All-received 或接收到对于所有文件片段的 ACK 后发送 Finish 报文并释放与此文件有关的资源。接收方接收到 Finish 报文后停止本文件的接收并释放相应的资源。

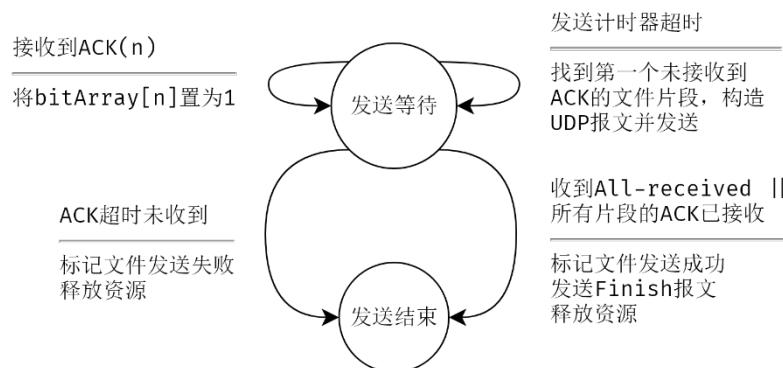


图 1.11 文件传输协议发送方状态机

文件发送方状态机如图 1.11 所示，发送方初始于发送等待状态。每当发送计时器超时后就寻找第一个未收到 ACK 的片段并进行发送并将发送计时器重置。发送计时器的超时间隔可以调节，用以控制文件发送速率。每当收到 ACK 后，将 bitArray 对应位置为 1 以表示此片断接收方已经收到。当收到来自接收方的 All-received 报文或所有文件片段的 ACK 已经收到时，向接收方发送 Finish 报文并释放发送此文件所需的资源。在整个过程中，若超过一定时间未收到

¹ <https://baike.baidu.com/item/%E4%B8%A4%E5%86%9B%E9%97%AE%E9%A2%98>

ACK，则告诉调用者文件发送失败，然后释放资源。

对于文件接受者而言，其状态机如图 1.12 所示。接收方处理完文件接收请求后，初始化于接收等待状态状态，此时已经初始化好大小与要接收文件大小相等的空文件。每当接收到一个文件片段时，将文件片段写入文件的对应位置后构造对应片段的 ACK 报文并发送。当所有文件片段接收完成后发送 All-received 报文并进入接收完成阶段，在这一阶段对于所有来自于发送方且与这个文件有关的报文均丢弃并发送 All-received 报文，直到收到 Finish 报文或超时未收到任何报文。这时告知调用者文件接收成功，释放相应资源，接收结束。在接收等待过程中若超时未接收道文件片段同样进入接收结束状态，不同的是此时需进行清零工作，包括删除未接收完成的文件以及释放相应的内存空间等，然后告知调用者文件接收失败。

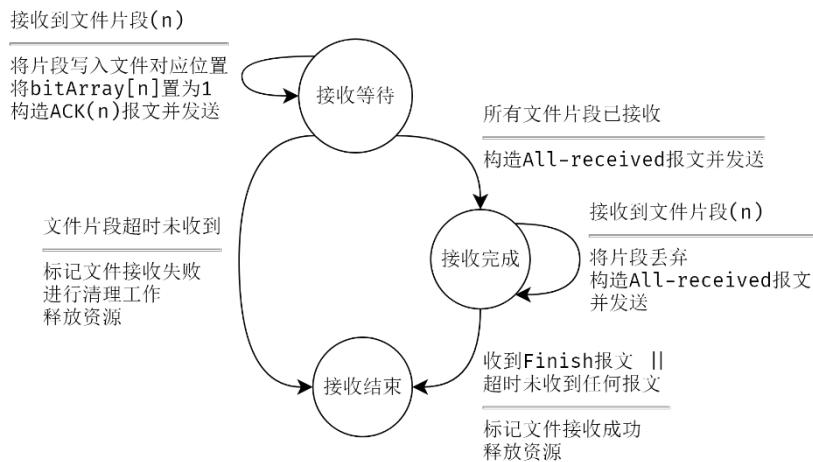


图 1.12 文件传输协议接收方状态机

1.4 系统实现

1.4.1 核心模块实现

核心模块包含了 3 个子模块（见 1.3.2 通信核心/协议部分架构），其中连接器模块和文件传输器模块完成了整个系统的主要的功能，是整个系统中最为核心的部分。整个核心模块以事件驱动的方式实现：TcpSocket 与 UdpSocket 均使用 Qt 封装的的异步 socket，采用非阻塞的方式进行，这样有利于同时处理多个 socket 的通信。

连接器实现

连接器类中包含的成员变量有 5 个 Map 类型的关联型容器，一个 QTcpServer 类型的监听用 TCP socket 和一个 QTimer 类型的心跳包发送定时器。在 5 个 Map 中，connectingMap, socketMap, addressPortMap, heartBeatTimerMap

这 4 个用于快速索引数据，其索引所用的键类型不尽相同，而所存储的均为 `SocketInfo` 类型的结构的指针，这些 `Map` 是连接器实现的核心与基础，余下的一个 `socketPool` 用于 TCP socket 的内存管理，不是本实现的重点，接下来不对其实行描述。

每一个 `SocketInfo` 实例存储了与一个对应 TCP socket 有关的数据，其中包括一个用于判定接收心跳包是否超时的定时器 `heartBeatReceiveTimer`，一个用于接受消息的缓冲区 `receiveBuf`，这个 Socket 所连接的对等端的 ip 地址和端口，一个用于存储对等端可能发出的应答的描述符的 `Map` 型容器 `pendingReceivMap`，以及将要发出的消息的队列 `pendingSendQueue`。

在上述的数据结构的基础上，连接器对外提供开始监听、尝试进行远程连接、尝试向远程发送消息以及断开远程连接这 4 个服务，这 4 个服务也提供了服务端与客户端通过 TCP socket 进行通信的全部功能。由于整个系统都是事件驱动的，因此在实现这些功能时需要考虑在任何情况下都不能发生阻塞（这也是实现中全部使用 Qt 所提供的非阻塞式 socket 的原因），正是由于这一点，在这 4 个服务中对于存在对事件循环而言不可忽略的时延的事件均采用回调函数的方式进行处理。接下来将先对这 4 个服务进行描述，然后对所有事件的回调函数进行统一的描述。

图 1.13 展示了开始监听服务的流程图，首先将监听 socket 接收到新连接的事件与 `newConnection` 方法绑定，用于处理监听成功后接收到的新连接，然后直接使监听用 socket 尝试开始监听并返回结果。使 socket 开始监听不会造成阻塞，而是会立即返回监听是否成功开始的结果，因此这一服务不会造成阻塞。

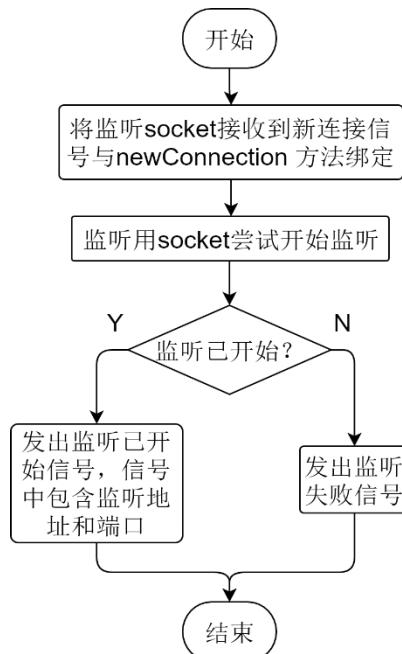


图 1.13 开始监听流程图

图 1.14 为尝试远程连接服务的流程图。在开始时首先判断所 3 尝试连接的地址和端口组成的键是否已经存在与 addressPortMap 中，如果已经存在说明对于这一地址/端口对的长连接已经建立，没有必要再次建立一个连接，因此直接发出“socket 已连接”信号并返回。当 addressPortMap 中不包含对应的地址/端口时，首先绑定 4 个回调方法，用于处理 socket 连接的成功与失败、以及 socket 连接后的数据处理。使用回调方法而不是直接在本方法中处理是由于 socket 连接是一个阻塞的过程，若直接在本方法中处理可能会造成阻塞。绑定

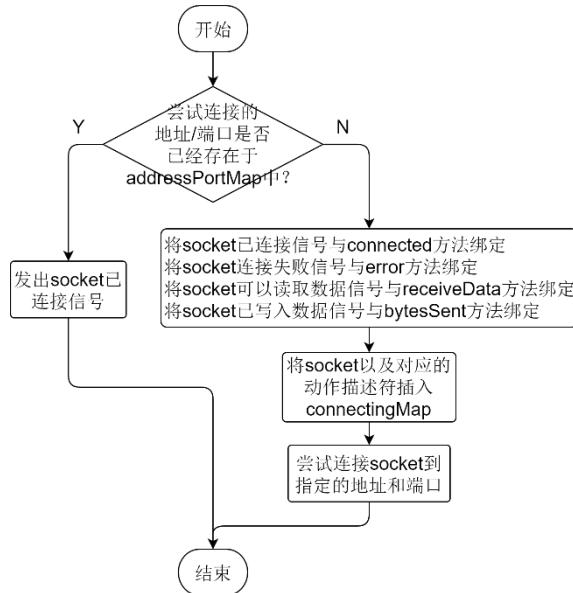


图 1.14 尝试远程连接服务流程图

完回调函数后首先将 socket 以及传入的动作描述符插入 connectingMap 中表示此 socket 正在进行连接过程，然后尝试开始连接指定的端口。

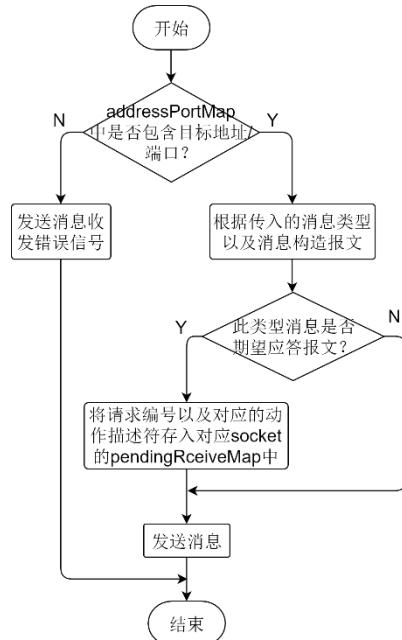


图 1.15 尝试发送消息流程图

图 1.15 描述了尝试发送消息的流程图。在发送消息之前，首先判断目标地址/端口是否已经存在于 addressPortMap 中（长连接是否已经建立），如果不存在，向调用者发送一个错误信号然后结束，否则通过判断传入的消息类型来构造报文，具体的构造方式见 1.3.5 小节。此外还要判断这一报文是否为请求报文，若为请求报文则应该有对应的应答报文，此时将对应的请求编号和动作描述符存入发送 socket 对应的 socketInfo 中的 pendingReceiveMap 中，以便应答报文到来时进行处理。在上述动作结束后，开始发送消息，在消息发送过程中如果出现错误则直接关闭 socket，释放相应的资源并通过信号的方式通知调用者。

断开远程连接这一过程则较为简单，首先查询需要断开的连接的地址和端口是否已经成功建立连接或正在连接中，若是，则直接将 socket 从所有对应的 Map 中移走，最后将 socket 从 socketPool 中删除以释放资源即可。

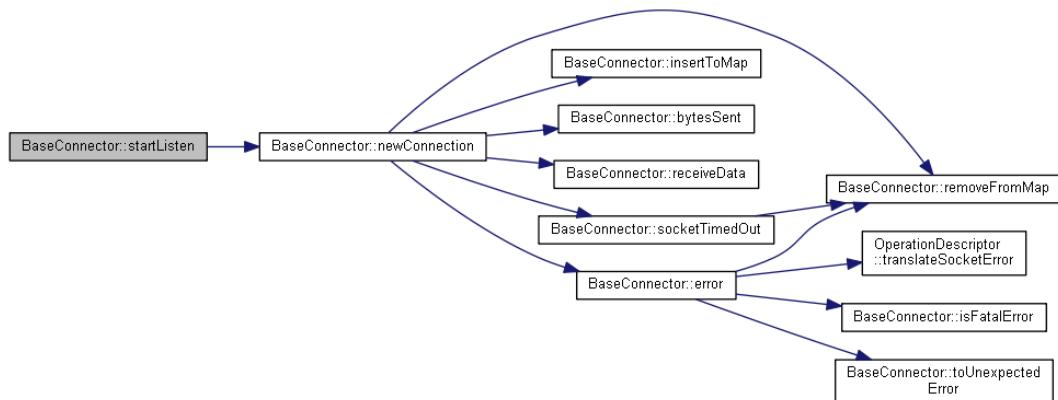


图 1.16 开始监听方法调用关系图

上述即为对于 4 个对外借口的总体描述。在这 4 个接口中，开始监听和尝试远程连接方法会绑定一些回调函数：开始监听方法会将 socket 发出的新连接信号与 `newConnection` 回调函数绑定，以便在有新的连接建立时进行处理，其在本类中的调用关系图（Call Graph）如图 1.16 所示。而尝试远程连接方法则会绑定 `connected`、`error`、`receiveData`、`bytesSent` 这 4 个回调函数，分别用于处理连接成功建立、有错误出现、有待接受消息以及消息部分或全部发送这中状

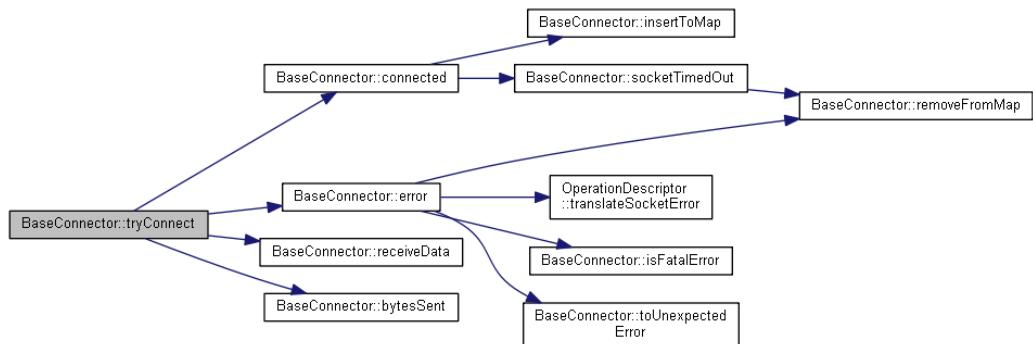


图 1.17 尝试远程连接调用关系图

况，他们均由 socket 所发出的信号触发。尝试远程连接方法在本类中的调用关系图如图 1.17 所示。

对于 newConnection 回调函数而言，它首先检测这一连接的地址和端口是否已经存在于 addressPortMap 中，如果已经存在说明连接已经建立，但对等端没有检测到这一连接因而建立了新的连接，此时调用 removeFromMap 将原来的 socket 从各个 Map 中移除并释放资源，然后使用新的连接代替旧的连接。若 addressPortMap 中不存在这一地址和端口，则直接加入各个 Map 中。在这一步骤处理完成后将 socket 的数据写入信号、有可读数据信号以及错误信号分别与 bytesWritten、readyRead、error 方法绑定，用于处理与 socket 有关的事件。

connected 回调函数用于处理主动发出的连接请求已经建立成功的事件，这一回调函数将 socket 从 connectingMap 中移出并加入到另外 3 个 Map 中，以表示连接已经成功建立，然后启动这一 socket 对应的 socketInfo 中的心跳包定时器，用于检测心跳包是否超时未收到。心跳包定时器的超时信号将与 socketTimedOut 函数绑定，当心跳包超时未收到时主动断开连接并释放对应的资源。

error 是一个较为重要的回调函数，它用于处理所有与 socket 有关的错误，无论是连接错误还是数据收发错误，其流程图如图 1.18 所示。在 Qt 的 socket 实现中，所有与 socket 有关的错误全部被封装在一个 error 信号中，而这些错误中有些是致命错误，有些是何以忽略的错误，剩下的则是不需要处理的错误。本类的 error 函数实现的功能是将底层 socket 发出的错误信号中的错误重新进行分类并处理或转发。对于 QTcpSocket 中的错误类型，首先判断其是否为致命

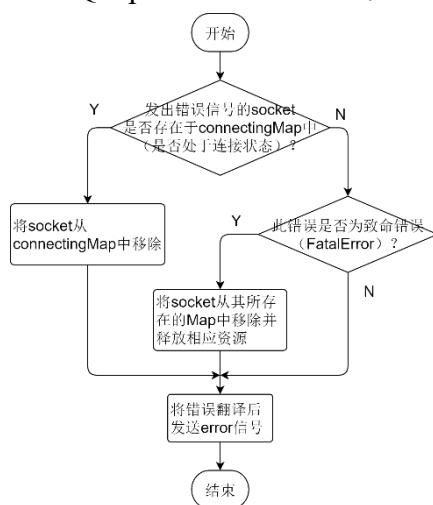


图 1.18 error 回调函数流程图

错误。若是，则将主动关闭 socket 以断开连接，然后释放与这个 socket 有关的资源，若为可恢复错误（如数据包过大无法发送），则清理与这一动作有关的资源并以将这一信号翻译为本类所使用的错误类型后再以信号的方式发送给调用者。

`receiveData` 则是用于处理数据接收的回调函数，此函数由 `socket` 接收到数据的信号驱动，其流程图如图 1.19 所示。由于 TCP 是面向流的传输协议，一次接收到的数据可能不完整，也可能包含两个报文的内容，因此需要 `receiveData` 这这样一个函数进行处理。在接收消息时，首先尝试接收一个完整的首部，在确保首部接收完整的情况下即可知道报文的总体长度，然后尝试接收整个报文。在这一过程中若尝试从 `socket` 中读取数据时缓冲区中没有内容则世界退出。此外 `receiveData` 还需将消息进行分类，分为期望的消息和意外的消息，其中期望的消息是指发出的请求所对应的应答，意外的消息是指对等端主动发出的请求。对于期望的消息，将之与发出的请求对应，然后通过动作描述符传递给调用者。对于意外的消息，通过 `unexpectedDataReceived` 信号发送给其它模块（由其他模块自行绑定这一信号）。

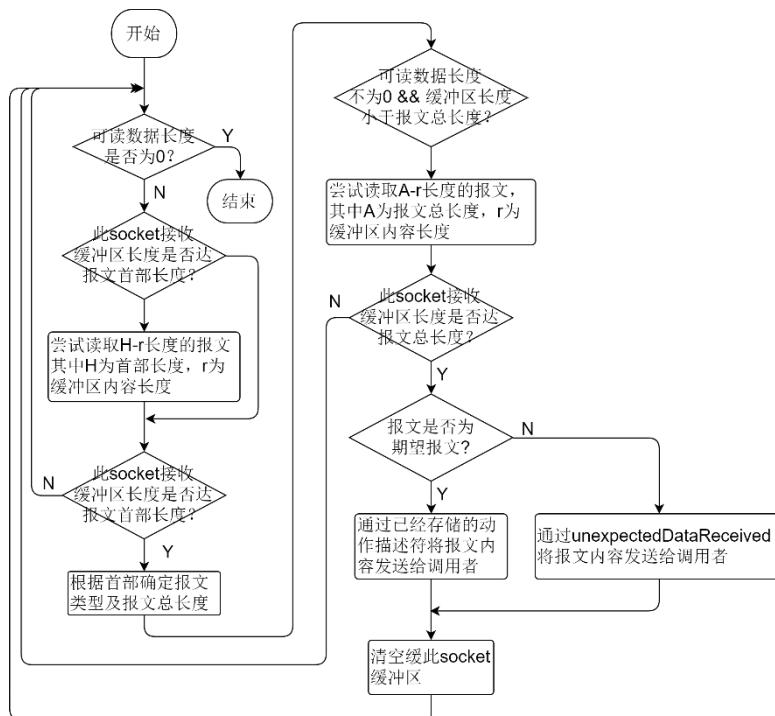


图 1.19 `receiveData` 回调函数流程图

`bytesSent` 回调方法则是用于处理消息发送队列。由于需要发送的消息的填充速度有可能比消息发送的速度快，因此一次不能将所有消息发送完，若强行尝试一次发送完所有的消息，那么由于消息发送过程需要较长时间，会发生阻塞，因此需要这样一个方法将几个报文之间的发送进行链式的连接：每当消息部分发送完时会触发这一回调函数，检查是否有下一个报文需要发送，由于 Qt 内部事件循环是使用队列处理的，因此两个报文发送之间可以穿插其它的事件，而不会造成阻塞。这一函数将消息发送队列中的报文视作一个流处理，每次尝试发送到一个报文结束的部分，然后等待进行下一个循环。

以上即是连接器的大体架构以及实现方法，总体而言，整个连接器类对外

提供开始监听、尝试进行远程连接、尝试向远程发送消息以及断开远程连接这 4 个服务，在内的内部所有的逻辑都是事件驱动的，4 个服务通过信号的方式驱动各个回调函数，各个回调函数再通过信号的方式直接或通过动作描述符间接地向外界传递请求和数据处理的结果。

文件传输器实现

文件传输器的主要功能是通过 UDP socket 传输文件。与连接器不同的是，文件传输器内部只有一个 UDP socket，不需要进行 socket 管理，这也是由于 UDP 不是基于流的协议所导致的。

文件传输器类内部除了包含一个用于文件收发的 UDP socket 之外，还包含 2 组共 6 个 Map 类型关联型容器，分别为 pendingSendMap、sendIdentifierMap、sendTimerMap、pendingReceigveMap、receiveIdentifierMap、receiveTimerMap，其中前三个用于存储正在发送的文件片段的信息，后三个用于存储正在接收的文件片段的信息。在这六个 map 中，由 pending 开头的用作 fileInfo 结构体的内存管理，其键值对中键为发送/接收文件的动作描述符，值为 fileInfo 类型的结构体。以 IdentifierMap 结尾的容器用于通过文件编号和发送方地址/端口来快速索引到 fileInfo 结构体，其键值对中键为对等端的 ip 地址、端口、发送/接收文件的文件编号所连接而成的字符串，值为相应的动作描述符。以 TimerMap 结尾的容器与前述容器功能类似，只是其索引用的键为指向 fileInfo 中的超时定时器的指针。

fileInfo 结构体用于存储与需要收发的文件相关的信息，其中包括 QFile 所定义的文件内的实例，一个 2 字节无符号整型的文件编号，对等段的 ip 地址和端口，一个 bitArray 类型的用于存放文件收发进度的数组，以及一个超时用定时器。

上述结构构成了文件传输器实现的基础，也是文件传输过程所直接操纵的对象。在这一基础上，文件传输器对外提供 sendFile 和 createPendingReceive 这两个服务。sendFile 用于通过文件传输器发送文件，而 createPendingReceive 用于告知文件传输器将有一个文件被接收。这两个服务只是用于初始化文件的收发，真正文件收发的服务的实现为两个回调函数 sendTimerTimedOut 和 processData，他们在文件传输器初始化的阶段就分别被绑定在了发送定时器超时事件和文件 socket 有可读数据事件上。

sendFile 的流程图如图 1.21 所示。sendFile 首先尝试打开文件，若打开文件失败则发送文件未成功打开信号给调用者然后返回。打开文件成功后检测文件大小是否超过协议所支持的最大大小（由于本协议使用 4byte 无符号整型来表示片段的顺序编号，每一个片段大小为常量 segmentSize，因此文件总大小不能超过 $2^{32} \times segmentSize$ 字节）。若文件过大则发送文件大小过大信号，释放

资源后返回。若能够通过此项检查，则初始化 fileInfo 结构体并将 fileInfo 和传入的动作描述符插入三个 Map 容器中，并开始 fileInfo 超时定时器，文件开始发送。

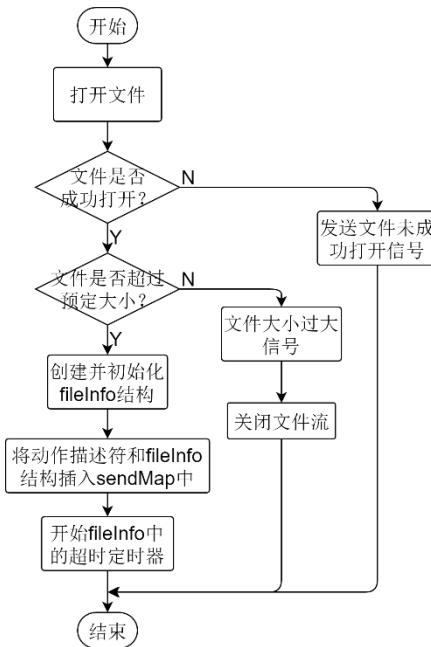


图 1.21 sendFile 流程图

createPendingReceive 的流程与 sendFile 的流程较为相似，其流程图如图 1.20 所示。它首先尝试初始化一个将要接收的文件大小的空文件用于保证文件不会在接收到一半时出现空间不足的情况，此大小在先前的文件传输沟通阶段已经从文件发送方得知。若不能创建这样的文件，那么就发送对应的信号然后

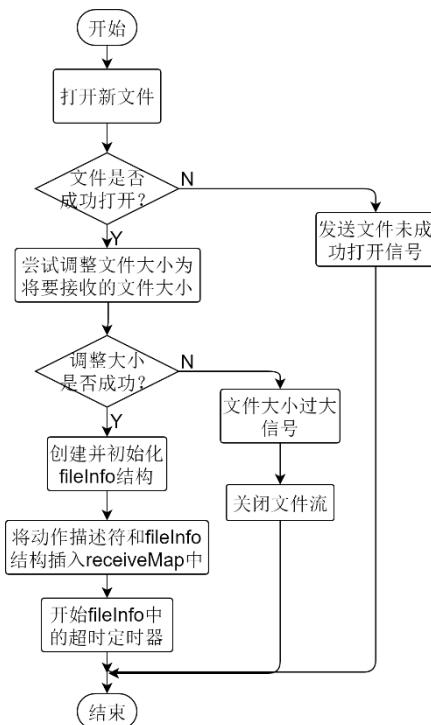


图 1.20 createPendingReceive 流程图

返回。如果文件成功创建，则直接将创建的 fileInfo 和对应的动作描述符插入 receiveMap 中，文件接收开始。

sendTimerTimedOut 回调函数再初始化阶段被绑定在 sendTimer 的超时事件上。sendTimer 为用于发送报文的定时器，其每隔一个较短的间隔超时一次，然后出发 sendTimerTimedOut 回调函数进行一次数据片段的发送。这样通过调整 sendTimer 的超时间隔就能够影响到发送速率和丢包率。sendTimerTimedOut 的流程图如图 1.22 所示。在每一次发送的过程中对于 pendingSendMap 中的每

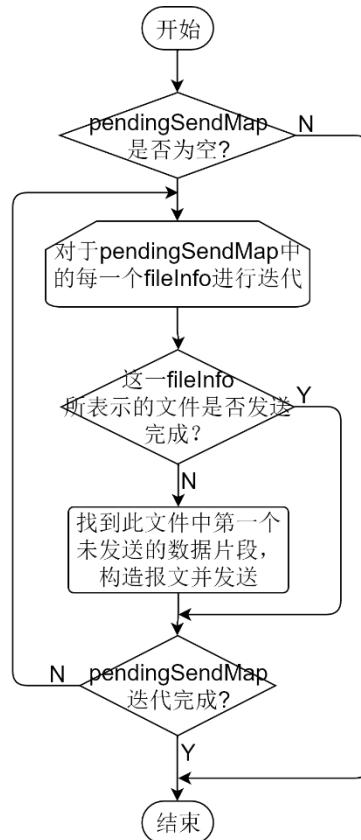


图 1.22 sendTimerTimedOut 流程图

一个文件进行迭代，对于每一个文件，若并不是所有片段都已经被标记为发送完成，那么寻找到其第一个没有被发送的片段进行发送。不使用滑动窗口（或者说直接将滑动窗口的大小设置为所有文件片段的总个数）是为了减少在丢包时停等的时间，由于本程序面向的使用者为局域网聊天用户，因此使用较为激进的做法以提高文件传输速率。sendTimerTimedOut 回调函数进行的工作较少，因为它只负责文件片段的发送。对于文件片段的接收和 ACK 的接收，则全部集中在 processData 回调函数中。

processData 需要对于 1.3.5 协议设计中提及的 ACK、All-received、Finished 报文以及文件数据片段进行处理，其流程图如图 1.23 所示。每当 socket 中有可读数据时此回调函数被触发。对于 socket 中所有的报文进行迭代，对于每一个报文，首先尝试进行解析并通过 crc32 校验和对报文进行校验，若报文解析

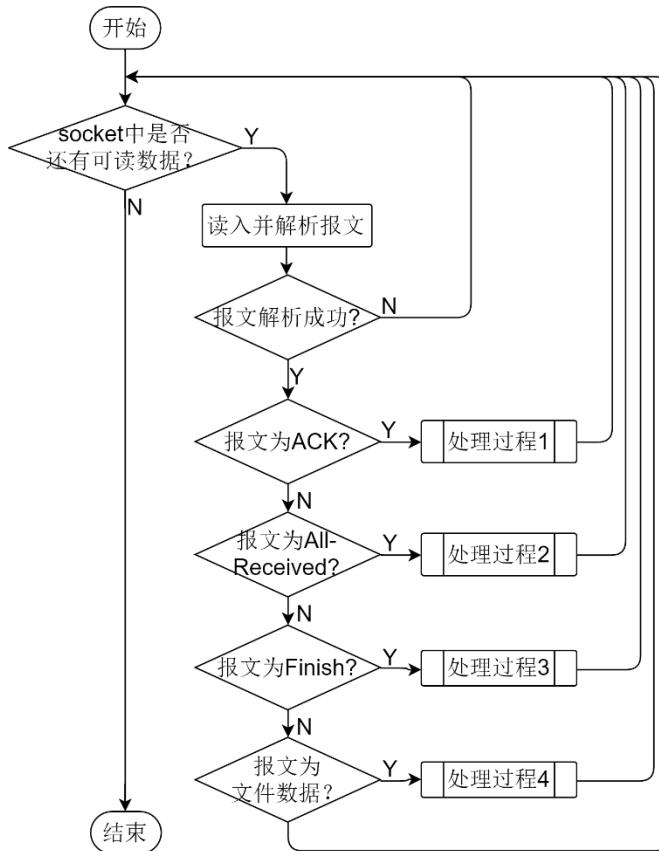


图 1.23 processData 流程图

失败或校验和校验失败，都视为报文解析不成功，丢弃当前报文并进行下一个循环。报文解析成功后判断报文的类型，根据协议的内容，可以被接收的一共有 4 种报文：文件片段数据报文 Data，文件片段已全部接收报文 All-received，结束发送报文 Finished 以及对于文件片段报文的确认报文 ACK。对于这 4 中不同的报文，判断后分别用与定义的过程处理。处理过程 1~4 的描述如下：

若接收到了 ACK 报文，说明本机为文件发送方，通过处理过程 1 进行处理。首先读入文件编号，根据文件编号在 pendingSendMap 中找出 fileInfo，然后将 fileInfo 中用于记录 ACK 接收进度的 bitArray 中对应位置的比特置为 1（无论这一位之前是否为 1）。然后统计 bitArray 中 1 的个数（Qt 所实现的 bitArray 提供了在 O(1) 时间内得到这一数值的方法），将其与总的要发送的文件片段数进行比较，若相等，说明文件已经完全被接收到。此时构造 Finished 报文并进行发送，通过动作描述符通知外部模块，然后将对应文件的 fileInfo 和动作描述符从 pendingSendMap 中移除，以释放对应的资源。

若收到了 All-received 报文，说明本机为文件发送方，通过处理过程 2 进行处理。且接收方已经接收到了所有的文件片段，此时直接构造对应的 Finished 报文并进行发送，然后通过存储于 pendingReceiveMap 中的动作描述符通知外部模块，最后将对应文件的 fileInfo 和动作描述符从 pendingSendMap 中移除，以释放对应的资源。

若接收到了 Finished 报文，说明本机为文件接收方，通过处理过程 3 进行处理。此时已经能够确定文件发送方主动停止了对文件的发送，直接通过动作描述符通知外部模块，将接收文件流关闭，将对应的 fileInfo 从 pendingReceiveMap 中移除以释放资源即可。

若接收到的为 Data 报文（文件片段），说明本机为文件接收方，通过处理过程 4 进行处理。去除此片段对于文件的偏移，然后检测对应的 fileInfo 中的 bitArray 中在这一偏移处是否为 1，若是，则说明这一片段已经写入文件，不用重复写入，直接构造对应文件对应片段的 ACK 报文进行发送即可。若此位为 0，说明这一片段未写入文件，首先将此位置为 1，然后将接收到的文件片段写入文件的对应位置。由于文件已经预先分配好大小，因此可以使用类似于随机访问的方式写入而不必每次都以追加的方式进行写入。首先测试 bitArray 中对应位是否为 1 而不是直接写入文件是因为文件 IO 的开销较大，对于重复发送的片段如果重复写入会造成内存及磁盘 IO 资源的浪费。在上述过程进行完后，类似于处理过程 1，直接统计对应 fileInfo 中 bitArray 中的 1 的个数，若其与将要接收的文件的总片段数相等，说明文件接收完成，构造对应文件的 All-received 报文进行发送，否则直接进行下一次对于 socket 中数据报文的读取。

除了上述 4 个过程之外，在任何一个阶段，若某一文件对应的 fileInfo 中的定时器超时，说明对于该文件而言超时未接收到任何报文，此时认为对等端故障，直接清理现场并释放对应的资源。

值得一提的是，在所有包文的构造过程中，校验所用的 crc32 都是以查表的方式进行构造的，这样可以极大的提升报文构造所需的时间。

动作描述符实现

实现动作描述符 operationDescriptor 是为了两个场景：动作区分和事件跟踪。由于 Qt 所实现的信号-槽模型是一种广播式的模型，即对于连接了一个信号的所有槽而言，每当这一信号被触发所有的槽都会被调用，这不利于不同动作的区分。对于这一问题，在 Qt 系统内部有不少解决方案，如使用过滤，将包含动作的信息以参数的形式附加在信号中，以及使用事件而不是信号等。但在所有的跨线程信息传输解决方案中这些要么会较大的降低程序的效率，要么会较多的增加无谓的代码。动作描述符的实现目的之一就是为了解决这一问题。动作描述符在被创建的时候就保证了每一个实例都是全局唯一的，并包含一个全局唯一的无符号 64 位整型 id 可供外部使用。

对于每个 id 唯一性的保证通过如下结构实现：动作描述符内中包含两个静态成员：idPool 以及 idNum。idPool 是一个队列型容器，用于存放能够被分配但未被分配的 id，idNum 则是一个 64 位无符号整型变量，用于存放当前已被分配和未被分配的 id 总数。id 分配从 0 开始，每次构造动作描述符时首先尝试

从 idPool 队列首中取出一个数作为自己的 id, 若 idPool 为空则说明所用已经构造的 id 都已经被分配, 此时使用 idNum 作为这一动作描述符的 id, 并将 idNum 加 1。这样就能保证每一个被分配的 id 都是独立的。在动作描述符析构时, 首先判断 id 是否为 idNum-1, 若是, 说明这是当前分配的所有 id 中最后一个, 此时对这一 id 进行回收: 直接将 idNum 减去 1, 否则将这一 id 加入 idPool 队列的尾部等待下一次使用。图 1.24 展示了这 4 种情况 (1、2 为构造, 3、4 为析构)。这样做保证了资源使用的最小化: 对于队列的操作次数以及对于队列长度的控制都降至最低。

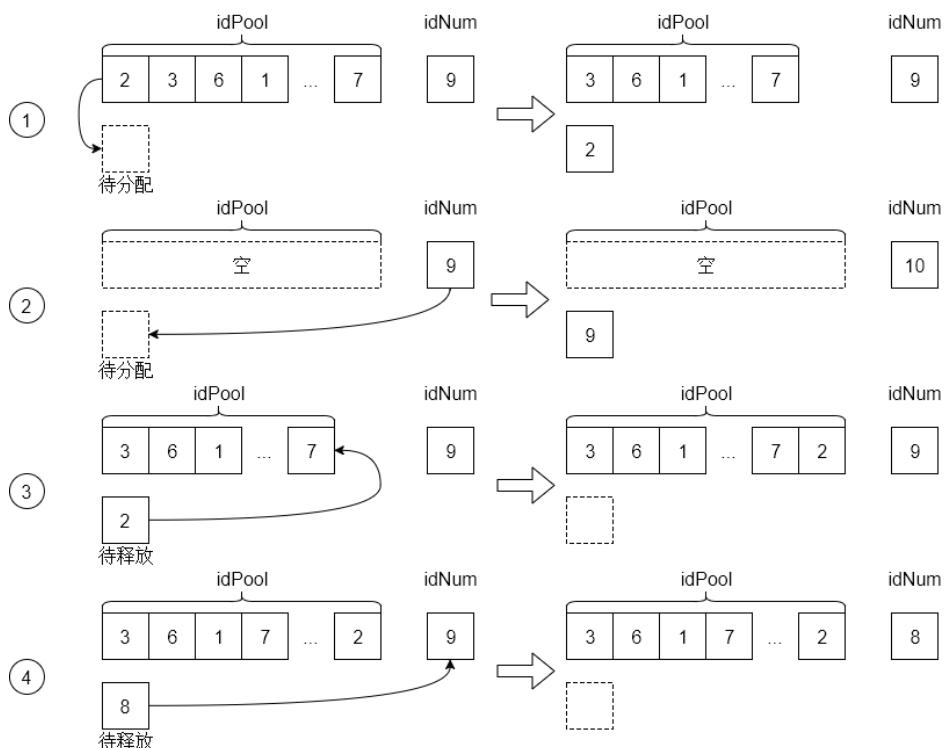


图 1.24 动作描述符 id 分配的 4 种情况

动作描述符的另一个功能则是事件的跟踪。这是通过对于信号过滤转发实现的。对于可能需要进行跟踪的动作, 在其函数参数中增加一个动作描述符, 函数需要通知外界的信号可以直接发送给这一描述符, 然后由描述符进行过滤并转发, 这样就避免了广播造成的混乱。动作描述符的默认参数可以设置为一个特殊的空描述符, 这样对于不需要跟踪的动作也不会强制要求传入一个参数。在有关联性动作存在时 (如发送请求报文和对于请求报文的接受, 以及发送分阶段报文的不同阶段等) 可以直接使用同一个描述符, 通过这种形式, 动作之间得以关联, 从而实现了外部模块在不影响内部模块的情况下对于事件的进展情况进行跟踪。

上述动作描述符与连接器, 文件传输器一同构成了通信核心模块。在客户端与服务端的实现中, 对于网络通信的需求均是通过动作核心模块来实现的。

1.4.2 客户端实现

客户端核心实现

客户端核心 ClientCore 用于沟通上层模块(图形用户界面)与底层模块(通信核心)，其主要的功能是在初始化时对对于客户端而言全局唯一的于核心模块进行初始化，并在 GUI 需要进行网络通信相关的操作时返回连接器或文件传输器的引用。此外，客户端核心还负责管理服务器的连接以及向外部模块发送服务器的连接状态。客户端核心类内部包含一个连接器，一个文件传输器，以及两个线程描述符，这两个线程分别用于运行连接器与文件传输器，充分利用计算机的多个核心，避免与 GUI 运行于同一个线程上。此外，客户端核心类的中存储了传输器的监听端口，文件连接器的收发端口，服务器的地址与端口，以便被调用时能够快速返回。

客户端核心对外界提供的服务有 getConnector, getFileTransceiver, 分别用于获得连接器和文件传输器的引用，connectToServer 用于连接服务器，isServerConnected, serverAddress 和 serverPort 则是用于获得与服务器有关的信息：服务器是否已经连接，服务器的 ip 地址以及服务器的端口等。这些服务的实现都极为简单，除了其名字所指示的功能外没有附加的操作。对于服务器的连接，在服务器连接完成后会将连接的结果通过信号的方式告知其他模块，当服务器连接意外中断时也会通过信号的方式进行告知。总体而言，ClientCore 是一个较薄的中间层，但是在整个客户端实现中时不可或缺的。

用户数据库的实现

用户端客户数据库用于临时存储在线用户的信息，当客户登录后服务端会推送这些信息到客户端，而登出后客户端则不会存储这些信息。用户数据库使用两个 Map 容器来存储用户信息：emailMap 和 addrPortMap，不像其它模块，用户数据库没有用于内存管理的容器，因为其所有容器的索引用键都不是指针型的，而存储的值只需使用智能指针便可以做到自动内存释放。Map 容器中存储的是 UserInfo 类型的结构，其中包含用户的邮箱（同时也是用户的唯一标识符）、用户名、用户的在线状态、用户的监听地址和端口、针对这一用户的所有消息传输记录和文件传输记录。

由于在客户端用户数据库直接由图形界面来操作，且其操作开销很小，因此其与图形用户界面存在于同一个线程上。对于外部提供增加、删除用户的接口 addUser 和 delUser，获得用户信息的接口 getUser 以及更改用户在线状态、添加消息记录、添加文件记录的接口 updateOnlineStatus、addMessageToLog、addFileToLog 等等。对于 getUser 方法有两个重载，分别用于通过用户邮箱获得用户信息和通过用户的 ip 地址和端口获得用户信息。

除了 updateOnlineStatus 方法外，其余的方法均为对于容器类操作以及异常

判断的简单封装，此处不给出其流程图。对于 `updateOnlineStatus` 而言，其流程图如图 1.25 所示。在更新网络状态之前，首先判断用户当前的状态以及新的状态。如果当前状态为在线，更新状态也为在线，说明更改的用户的网络环境

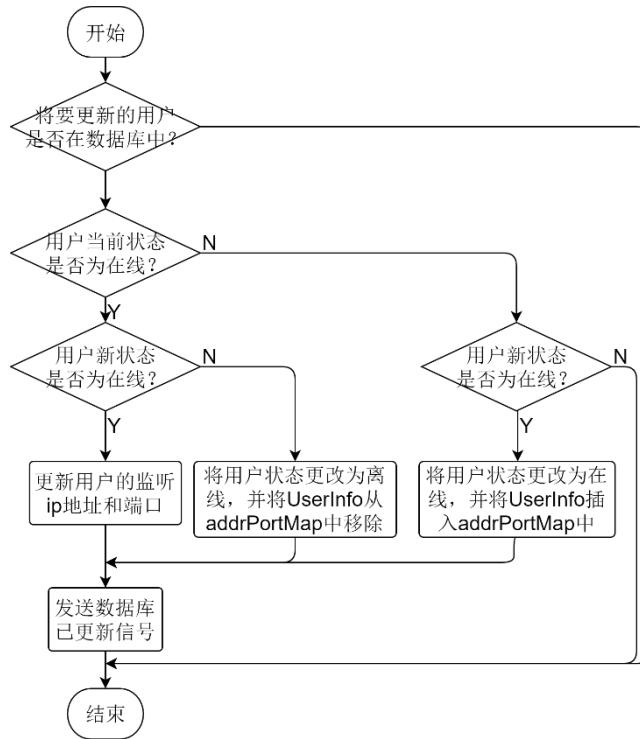


图 1.25 `updateOnlineStatus` 流程图

发生了变化，此时需要更新用户的监听地址和端口。若当前状态为离线，更新状态也为离线，那么将这一用户的状态更改为离线，并从 `addrPortMap` 中移除。若用户状态从离线转换为在线，那么将其状态更改为在线并插入 `addrPortMap` 中。若更改前后状态均为离线，则无需进行任何更改。在前三种更改发生后，发送一个数据库以更新信号以告知外界数据库已经更新。事实上，在本数据库的实现中，任何实质上更改了数据库的方法都会触发数据库已更新信号，这一信号将在需要显示用户信息的地方被使用。

图形用户界面实现

图形用户界面实现的大部分内容是界面的设计、信号绑定以及对于收发消息的处理。由于其模块众多且大多实现简单，下面只展示其界面并对重要的方法进行描述。

在用户启动程序时，首先看到的是登录界面，在此之前，需要对于整个程序进行初始化，即为用户核心的创建、用户数据库的创建以及二者的初始化。在初始化完成后才显示登录界面，以保证用户在初始化之前不能进行操作，登录界面如图 1.26 所示。对于图中的每一个按钮都有一个与之绑定的对应的函数，当按钮被按下时函数将被调用。其中，选项按钮被按下将显示选项对话框，用户注册按钮被按下将显示用户注册对话框，密码找回按钮被按下将显示密码

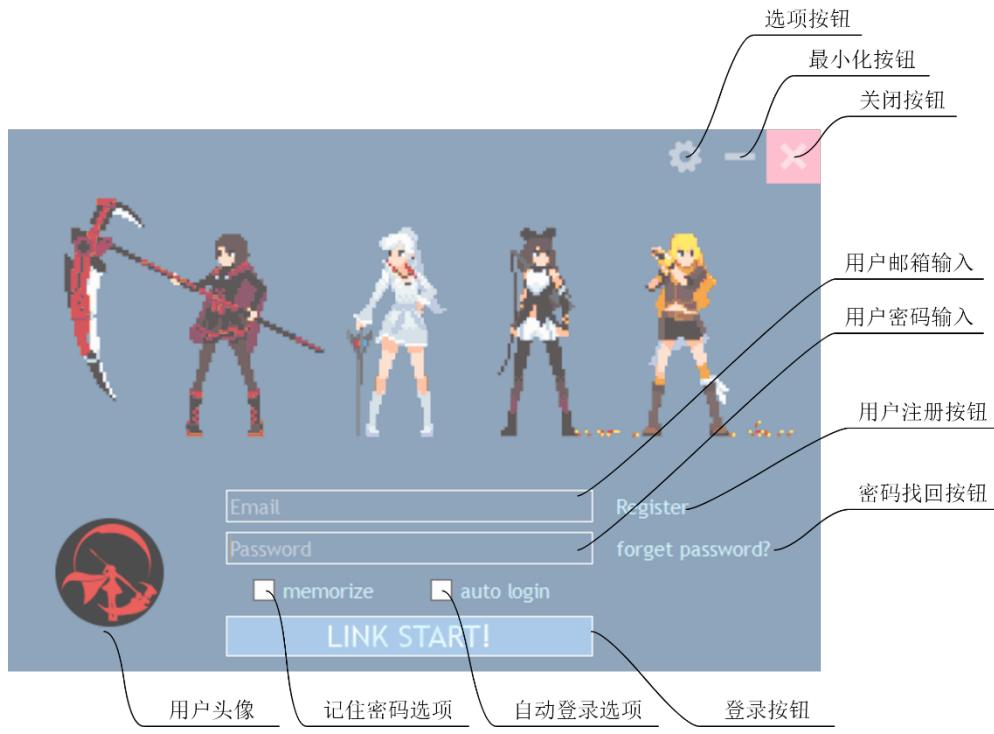


图 1.26 登录界面

找回对话框，而登录按钮被按下则将尝试进行登录。除了登录按钮外，另外三个按钮被按下都是直接构造并显示对应的对话框。登录按钮被按下时，则是首先通过客户端核心确定服务器是否已经连接，若未连接，则显示一则错误信息提示用户已未连接服务器。若已经连接，则读取用户邮箱输入和用户密码输入中的内容，按照协议构造 Json 结构，并通过客户端核心获得的连接器将数据发送。在发送数据前，一个 pendingDataReceived 回调函数被绑定在动作描述符的收到消息信号上，通过 pendingDataReceived 来处理由服务器返回的消息。如果返回的消息为登录成功，则从消息中读取当前在线用户的信息，然后将这些信息添加到用户数据库中，在添加完成后显示主界面。

选项对话框如图 1.27 所示。其中包含服务器地址、服务器端口输入框、一个连接按钮以及一个消息显示区（Connect 按钮下方的空白区域）。由于不像商业聊天软件或大型开源聊天软件一样有固定的域名或 ip，本聊天软件需要用户在登录之前手工指定服务器的 ip 地址以及端口。在服务端的实现中，服务器的

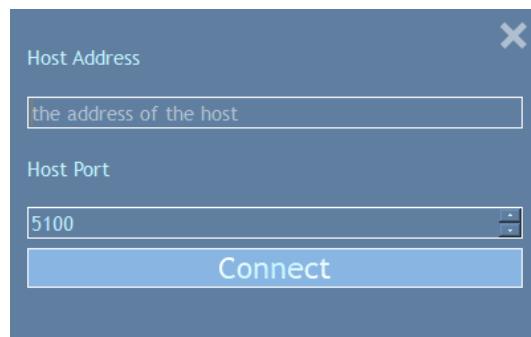


图 1.27 选项对话框

端口默认为 5100，因此在此处端口默认值为 5100(可修改)。当用户点击 Connect 按钮时，回调函数被触发，尝试通过连接器来连接服务器。在连接过程中，所有的交互式部件被禁用直到服务器连接成功或连接失败，若连接失败，将在消息显示区显示失败的原因，若连接成功则会直接退出本对话框。

用户注册对话框在登录界面中的用户注册按钮被按下后弹出，对话框包含 5 个输入框、两个按钮以及一个状态指示区(最下方的空白部分)，如图 1.28 所示。五个输入框从上到下分别为：用户邮箱输入框，用户名输入框，密码输入框，安全问题输入框以及安全问题答案输入框，其名字即指示了输入的内容。右侧的两个按钮分别为用户头像选择按钮(Avatar)和注册按钮(Register)。当注册按钮被按下时，调用的回调函数会读取这些输入框中以及用户图标的信息，将其构造为 Json 数据并尝试发送给服务端。若注册成功则会在状态指示区显示注册成功的文字，否则会显示对应的错误信息。

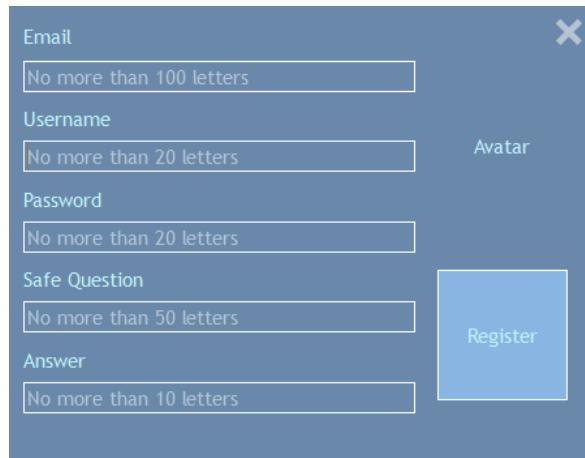


图 1.28 用户注册对话框

若用户忘记了密码，可以在登录界面点击找回密码按钮(forget password? 按钮)来找回密码。密码找回对话框如图 1.29 所示。其包含 4 个输入框，一个提交按钮和一个状态显示区(提交按钮的下方)。4 个输入框自上而下分别为用户邮箱，安全问题，安全问题答案，新密码以及重复新密码输入框。在开始时，

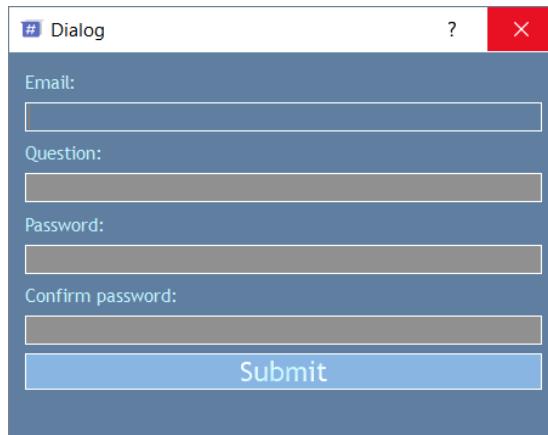


图 1.29 找回密码对话框

只有用户悠闲对话框是激活（可以输入）的。在用户输入邮箱并点击提交按钮后，回调函数构造 Json 数据并发送给服务器。在服务器返回结果且结果为通过的情况下，标签 Question 后会出现实现存储的问题，并且此时只有第二个输入框可以使用。在输入框输入答案后再次点击 Submit 将数据发送给服务器。服务器接受答案并且验证通过，返回通过报文后第 1、2 个输入框被禁用，第 3、4 个输入框被激活。此时输入两遍新密码，第三次点击 Submit 后将数据发送给服务器，在一切正常的情况下密码将更新成功，在状态显示区会提示密码更新成功。上述即为对于协议中找回密码三个阶段的实现。在此过程中任何部分出现问题（如用户不存在，答案错误，超时等）将会在状态显示区显示相应的错误并退回到第一阶段。

在登录界面成功登录后，会显示主界面。主界面分为 3 栏：用户栏，消息栏以及文件栏。在登录界面用户列表被取回后，其中的用户会显示在用户面板中。选择对应的用户就可以在消息面板中看到当前用于与这一用户的消息记录，

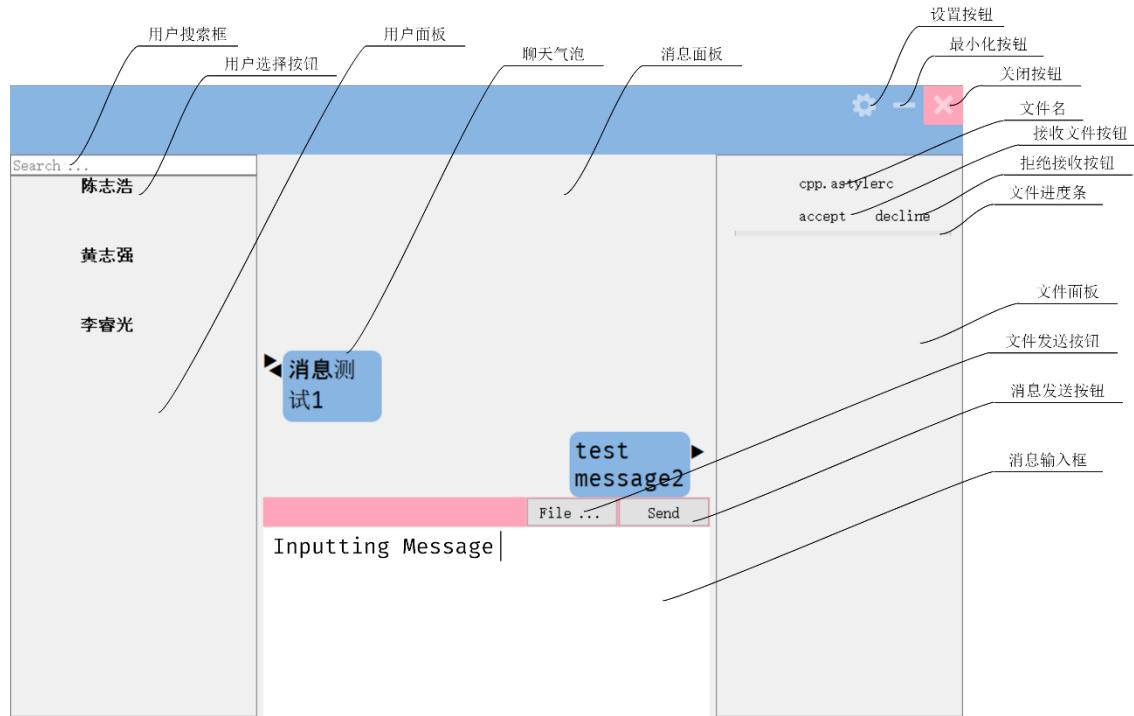


图 1.30 主界面

以及离线缓存的消息记录。这一过程在 userInfoPressed 回调函数中完成，它从数据库中读取消息记录，构造消息面板并用构造的消息面板替换当前的面板。

值得一提的是消息发送的过程和文件发送的过程。当 send 按钮被按下后，触发 messageButtonPressed 回调函数，但消息发送过程并不是立即在这个回调函数中执行。由于通信核心模块只负责消息的收发和 socket 管理，不负责用户层面的逻辑判断，因此在发送消息需要分为多个阶段进行，又由于不能阻塞 GUI 线程，各个阶段的处理函数之间必须使用信号的方式进行连接。消息发送

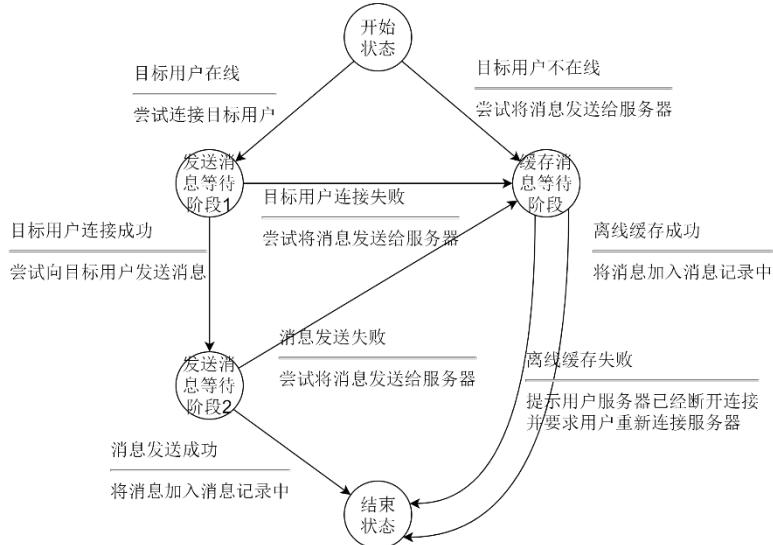


图 1.31 发送消息状态转移图

的状态转移图如图 1.31 所示。在发送消息前 `messageButtonPressed` 函数通过读取数据库中的用户信息判断用户是否在线，然后根据这一在线状况判断是直接点对点将消息发送给用户还是发送给服务器进行缓存。若用户在线则进行连接——发送两个阶段，若用户不在线则直接将消息发送给服务器。在连接——发送过程中有任何一处失败则直接转而将消息发送给服务器进行缓存，也就是说服务器缓存是作为整个消息发送过程最后手段。如果服务器缓存不能成功，意味着用户与服务器已经断开连接（已掉线），此时提示用户掉线并要求用户重新连接服务器。尝试连接目标用户的过程在连接器中做了优化：若用户已经连接，则会直接发送已连接而不会重新进行连接。

在这个过程中，由开始状态向发送消息等待阶段或缓存消息等待阶段进行的转移是通过 `messageButtonPressed` 函数实现的。从发送消息等待阶段 1 开始的状态转移和命令执行是由 `messageStage1Connected` 和 `messageStage1-ConnectFailed` 实现的，分别为连接成功的处理函数和连接失败的处理函数。由发送消息等待阶段 2 进行的转移是由 `messageStage2DataSent` 和 `messageStage2-FailedToSend` 实现的，分别用于处理发送成功和发送失败的情况。从缓存消息等待阶段进行的成功转移是由 `messageCachedStageReplied` 实现的，而失败转移则部分复用了 `messagesStage2FailedToSend` 回调函数。

文件发送的过程与消息发送的过程类似，但没有将文件缓存到服务器的功能因此也没有缓存文件等待状态，此外用于文件发送前首先要进行一次沟通，得知双方的文件发送端口、文件名以及文件大小等信息，因此比消息发送要多一个阶段。其状态转移图如图 1.32 所示。从开始阶段到文件发送完成中间共有三个状态四个过程，这四个过程分别用于连接目标用户，进行文件传输前的沟通，开始文件传输以及文件传输完成。除了从开始状态进行转移的过程使用 `fileButtonPressed` 实现以外，其它 6 个过程均使用回调函数 `fileStage<X><State>`

回调函数进行处理其中<X>为转移开始前的等待状态编号而<State>为触发这一转移的事件状态(如Failed)。在文件发送过程中任一时刻出现文件不能被传输的问题则立刻通知哦用户，释放已经分配的资源然后结束文件的传输。

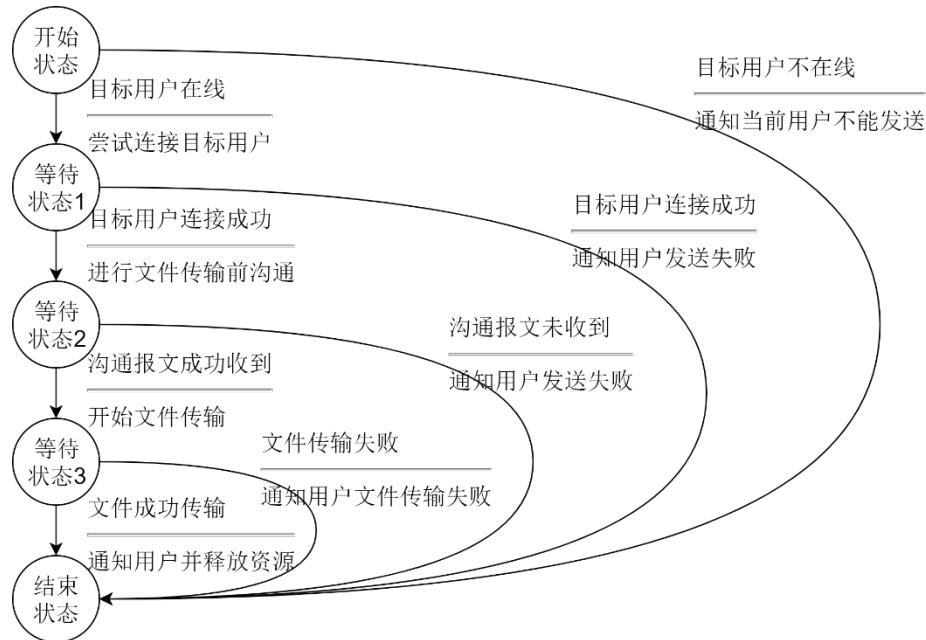


图 1.32 文件传输状态转移图

在文件传输开始前，动作描述符上的 `progressChaged` 信号会被绑定在 GUI 文件显示模块对应的槽中，这样在文件传输时即可实时显示传输的进度和速度。

上述几个模块即为图形用于界面的核心实现，除了这几个模块之外，还有另外的 `ClickableLabel`、`Animation` 等多个模块用于配合核心模块进行显示工作，由于这些模块实现细节并非重点且过于琐碎，在此不再赘述。

1.4.3 服务端实现

服务端的实现相较于客户端有许多共同点，如它们都包含一个核心模块，一个用户数据库以及多个模块组成的图形界面模块等，但由于服务器本身特性，又有一些不同：如考虑到服务器需要同时处理来自多个客户端的请求，将数据处理的逻辑封装到了 `DataProcessor` 类中进行处理，而弱化了图形用户界面的逻辑，数据库实现了持久化以及加密的功能等等。

服务端核心实现

相较于客户端核心，服务端核心进行的工作极少：只用于初始化通信核心（包括使消息端口开始监听），连接核心的信号与数据处理器中的部分方法以及初始化图形界面，这一过程会在服务端核心构造的被时候调用。由于服务端以数据处理的功能为主，用户界面可有可无，因此使用服务端核心启动用户界面而不是像客户端那样使用用户界面来初始化客户端核心。由于不需要提供离线文件的功能，服务端核心模块对外提供的接口仅有 `getConnector` 一个，用于

取出被服务端核心初始化的连接器单例，所有消息的收发均在返回的连接器上进行操作。

数据处理器实现

数据处理器是整个服务端最重要、最为核心的部分，所有来自于客户端的请求都要通过数据处理器进行处理，然后进行对应的应答。数据处理器中包含一个 `operationPool` 和一个 `operationMap`，前者用于动作描述符的内存管理，后者则是以动作描述符为键，`{数据类型, 数据内容}` 键值对为值的容器，这一容器用于临时存放请求的内容。服务器在接收到一个请求并处理之后，不能保证处理的结果一定能够成功发送给请求发送方，因此需要确认后再将结果发送，否则可能造成双方状态不一致（服务端已经处理了请求并写入了数据库但是客户端认为处理失败）的情况。由于不能阻塞够等待消息发送成功，因此需要一个回调函数来处理消息发送成功的事件，而这一 `operationMap` 这一容器就是在两个函数之间传递数据的载体。

此外，数据处理器中还有一个 `validateMap` 和一个 `validateTimeMap`，用于处理多阶段的请求（如密码找回请求），确保多个请求为同一客户端发出，请求之间的超时间隔不大于一个固定值以避免长时间无响应导致的资源泄漏。

每当客户端有请求到来时，`processData` 方法被触发，这一方法与 `dataSent` 回调函数一同组成了对于请求处理的核心部件：`processData` 用于解析、处理、构造并发送回报文，而 `dataSend` 负责回报文被确认发送后的收尾工作（如确认请求处理结果，写入数据库等）。整个数据的处理流程如图 1.33 所示。途中，根据报文类型做出相应处理是由一个 `switch-case` 语句块构成，选择项为协议中定义的报文类型。对于所有的请求，首先判断请求发送方是否存在与数据库中，如果数据库中不存在说明用户未在此服务端注册，拒绝提供服务并返回

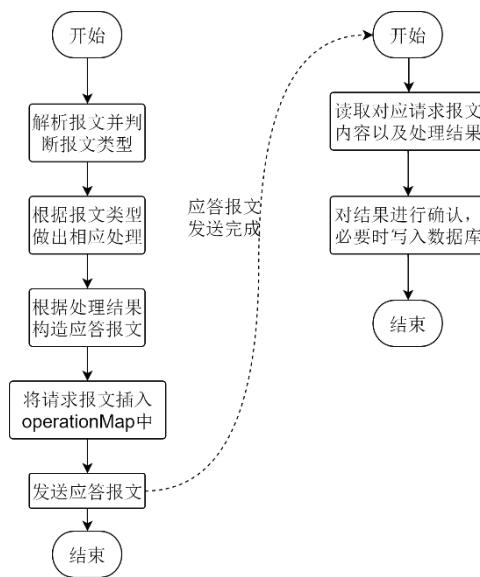


图 1.33 报文处理流程

拒绝提供回应。这样的请求不需要加入 operationMap 中，因为不需要关心回应报文是否能够发送成功。如果用户存在，则首先判定请求能否成功处理：对于注册请求，判断用户提供的信息是否完整且合法；对于登录请求，判断用户密码是否正确，对于更改密码请求的第二阶段，判断其提供的回答是否正确，对于消息缓存请求，判断目标用户是否存在。无论是否通过检测，都需要构造对应的相应报文进行发送。在确认消息已经成功发送后，dataSend 回调函数会将预先存储的请求处理结果写入数据库。

数据库实现

服务端的数据库与客户端数据库的实现类似，但提供了更多的功能，包括序列化与持久化的功能以及加密的功能。在服务端数据库类 DataBase 中，存在 3 个 Map 型容器：emailMap，onlineMap，onlineAddressPortMap。他们的键类型都为 string，存储的值均为 UserInfo 类型的结构体。emailMap 用于通过用户邮箱来索引用户信息，这个容器中存储了所有已经在这个服务器注册过的用户。由于用户名可能重叠而每个用户邮箱一定是唯一的，因此在设计中就采用了使用邮箱来索引用户的机制。onlineMap 只存储当前在线用户的信息，并且其使用用户的 ip 地址和监听端口进行连接得到的字符串作为索引，注意此处为监听端口而不是连接端口，每个客户端用于监听的端口和用于连接服务器的端口是两个不同的端口且均为随机分配的。onlineAddressPortMap 也只存储当前在线用户的信息，并使用用户的 ip 地址和连接服务器所用端口进行连接得到的字符串作为键。onlineMap 主要用于客户端之间的，需要经过服务器的通信，因为客户端之间使用的端口均为其监听端口，而服务器从不主动连接客户端，因此通过使用 onlineAddressPortMap，使用客户端连接服务器的端口来索引需要通信的客户端。

上述三个 Map 型容器存储的 UserInfo 结构体用于描述用户的所有信息。其中静态数据包括用户邮箱、用户名、Hash 过后的密码，用于密码找回的安全问题以及 Hash 过的答案。在运行时动态变化的数据包括客户端是否在线、客户端的 ip 地址、监听端口以及用于连接服务器的端口，其它用户发送给当前用户的离线消息列表。离线消息列表为一队列型容器，容器值类型为 {string, string} 类型的键值对，存储的为向此用户发送消息的用户的邮箱以及消息的内容。

除上述结构之外，数据库类中还包含一个互斥锁，用于多线程操作数据库时保证安全。虽然目前来看只有 dataProcessor 一个模块在操作数据库，但考虑到负载问题以及可扩展性，加上互斥锁在多个线程上的多个模块同时操控数据库时依然可以保证安全。

客户端实现中对于数据库的实现在服务端已经全部包含，其中服务端的为用户缓存消息服务 cacheMessage 与客户端数据库中的 addMessageToLog 类似，

其余均相同，此处不再赘述（见 1.4.2 客户端实现）。下面只描述其增加的部分：`serialize` 和 `deSerialize`，分别用于存储数据到磁盘以及从磁盘中读取数据。

对于为了数据存取的方便，未加密的数据首先转换为 Json 格式明文，进行加密后存储于单个问价夹的单个文件中，对于每一个用于有一个与其用户邮箱相同的文件夹，在文件夹中有一个名为`_info` 的文件，用于存储这一数据。这样设计也是出于可扩展性的考虑：若需要增加离线文件缓存功能，使用多个文件夹有助于区分不同用户的同名文件，数据文件夹的一个例子如图 1.34 所示。

```
tree datas/
datas/
├── chenzhihao1@hotmail.com
│   └── _info
├── huangzhiqiang1@hotmail.com
│   └── _info
├── husixu1@hotmail.com
│   └── _info
└── liruiguang1@hotmail.com
    └── _info
```

图 1.34 数据文件夹结构

序列化的流程图如图 1.35 所示。首先尝试进入或创建并进入文件夹，如果失败则直接处理下一个用户，进入文件夹以后读取一个用户的 UserInfo，转换为 Json 格式后进行加密，然后尝试写入`_info` 文件，如果不能写入亦处理下一个用户，直到用户全部处理完成。对于出错处发送对应的错误信号以告知调用者。

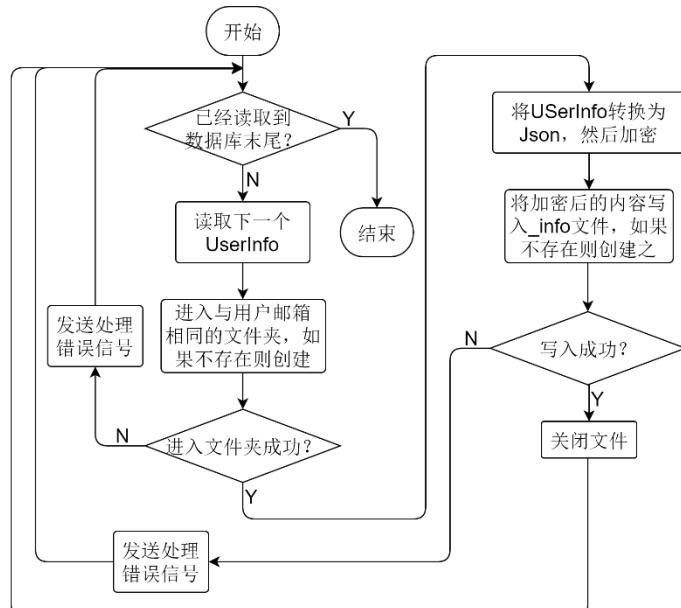


图 1.35 序列化流程图

反序列化的过程与序列化类似，首先进入各个文件夹并尝试读取`_info` 文件，若`_info` 文件读取错误（不存在或权限问题）则认为此用户不存在，读取出

来的数据首先解密得到 Json 明文然后再转换为 UserInfo 结构体即可。在序列化与反序列化的过程中，使用的加密方式为 AES 加密，使用的密钥为一个用于事先输入的密码经过 MD5 Hash 变换后得到的密钥，此密码能够在图形界面中进行更改。

图形用户界面实现

在服务端中，图形界面仅执行两个功能：数据库显示以及数据库密码更改。对于每一次数据库更新，图形界面都需要进行一次刷新以保证数据为最新的。图形界面包括主窗口，密码更改模块和密码验证模块这三个模块构成。

主模块主窗口如图 1.36 所示。当点击数据库界面中的任一栏时，主界面回调函数 userClicked 会读取数据库中的 UserInfo 并将其中的键值对一一显示在详细用户信息键值对中。updateTimedOut 回调函数使用了一个定时器来保证主界面刷新频率不超过一个固定值（如 10Hz），在人眼无法察觉的前提下用于应对服务器负载过重时数据库频繁更新的情况，以减小频繁更新界面造成的资源的浪费。validatePassword 按钮用于验证预设的密码，这一密码经过 MD5 Hash 后将用于数据库加密的 AES 密钥，changePasswd 按钮则是用于初始化或者修改这一密码。在主界面初始时，单击 validatePassword 并验证密码后才会加载已经加密过后的数据，在主界面停止运行前会自动将内存中的所有数据序列化并存储于磁盘上。

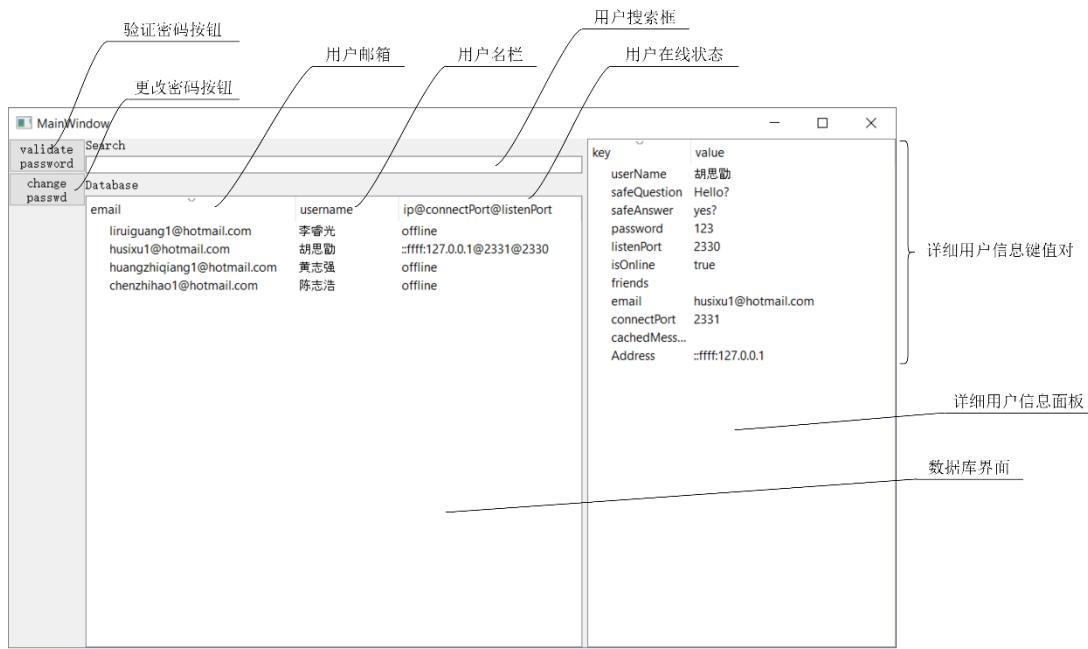


图 1.36 服务器主界面

验证密码对话框与更新密码对话框如图 1.37 所示，他们分别在 validatePassword 和 changePasswd 按钮被按下后弹出。其回调函数实现的逻辑均十分简单，验证对话框即将输入的密码进行 Hash 后与预先存储的 Hash 进行对比，然后判断是否一致，若一致则进行数据库解密并加载其内容，若不一致

则拒绝加载数据库内容。而更改密码对话框要求输入 1 遍旧密码（初始化密码时此输入框为禁用状态）与两遍新密码，验证旧密码通过后使用新密码的 MD5 值替换旧密码即可。

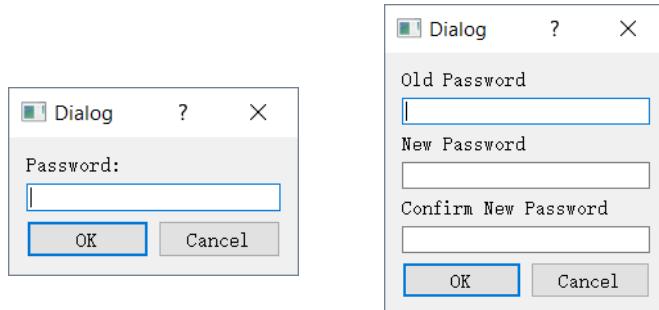


图 1.37 验证密码与更新密码对话框

1.5 系统测试及结果说明

1.5.1 测试环境

处理器： Intel® Core™ i5-4570 CPU @ 3.20GHz 3.20GHz
内存大小： 8.00 GB (7.89GB 可用)
操作系统： Windows 7 (7.0) 64 位 6.1.7601 Service Pack 1
网卡速率上限： 1Gbps
IP 协议版本： IPv6

1.5.2 功能测试

功能测试主要测试用户注册、用户登录、密码找回、在线聊天、离线消息、文件传输、数据库加解密以及多组用户同时聊天的功能。在进行所有的功能测试之前，都默认服务器是已经连接的状态。

用户注册测试

在没有用户注册时，数据库为空。然后在客户端进行注册。首先注册一个邮箱为 test1@hotmail.com，用户名为 Test1，密码为 123456，安全问题为“abc”，答案为 123 的用户。注册时的界面如图 1.38 所示。此时服务端界面中可以看到一个名为 Test1 的用户。按照上述方法再次注册用户名分别为 Test2、Test3、Test4 的用户，邮箱分别为 test2@hotmail.com、test3@hotmail.com、test4@hotmail.com，为了方便起见，其密码、安全问题与安全问题的答案与 Test1 用户一致。注册成功后，服务端界面如图 1.39 所示，其右半部分为 Test1 的详细信息。由于打开了调试模式，在服务端能够看到用户的密码，便于观察。在正常情况下，服务端是不能观察到明文密码的，其存储的为哈希过后的密码。

从图中可以看出，注册测试正常，客户端能够正确地将测试信息传递到服务端并且服务端能够正常处理这些信息。

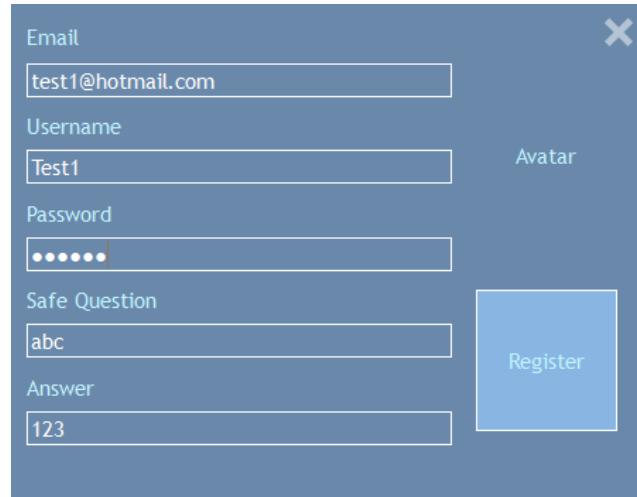


图 1.38 注册界面

MainWindow		
validate password	Search	
change passwd	Database	
	email	username
	test4@hotmail.com	Test4
	test3@hotmail.com	Test3
	test2@hotmail.com	Test2
	test1@hotmail.com	Test1

key value

key	value
userNmae	Test1
safeQuestion	abc
safeAnswer	123
password	123456
listenPort	0
isOnline	false
friends	
email	test1@hotmail.com
connectPort	0
cachedMess...	
Address	

图 1.39 注册后服务端界面

用户登录测试

现对于注册测试中注册的用户 Test1 进行登录测试。在登录界面的用户名输入框中输入 test1@hotmail.com，密码输入框中输入 123456 并点击登录按钮，能够正常弹出登录界面，并显示另外三个用户，如图 1.40 所示。而此时服务端的显示如图 1.41，能够正确的显示登录后 Test1 的在线状态、ip 地址、连接端口以及监听端口。说明登录测试正常，用户能够正确的进行登录。

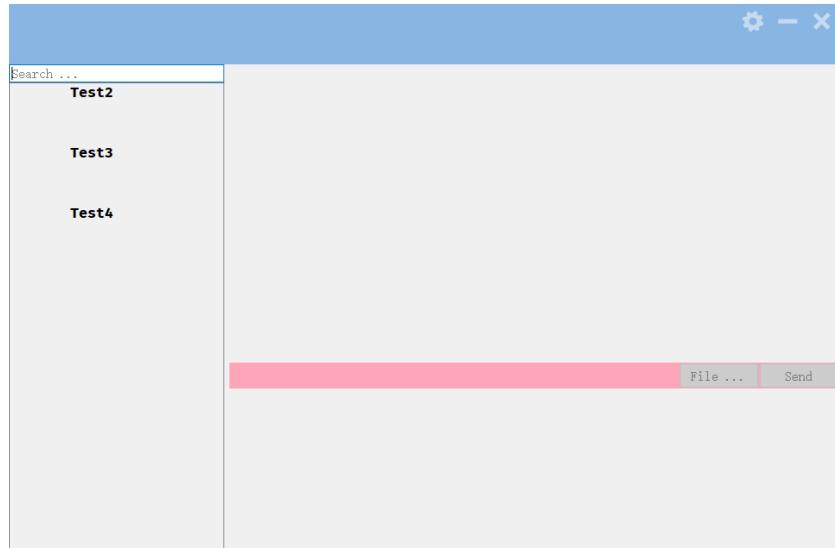


图 1.40 登录后主界面

A screenshot of a Windows-style application window titled 'MainWindow'. The left side features a sidebar with tabs for 'validate password' (selected), 'change passwd', and 'Database'. The 'Database' tab is active, displaying a table with columns 'email', 'username', and 'ip@connectPort@listenPort'. The table contains five rows: 'test4@hotmail.com Test4 offline', 'test3@hotmail.com Test3 offline', 'test2@hotmail.com Test2 offline', 'test1@hotmail.com Test1 ::ffff:127.0.0.1@6974@6971'. To the right of the table is a configuration table with 'key' and 'value' columns. The data includes: key 'userName' value 'Test1', key 'safeQuestion' value 'abc', key 'safeAnswer' value '123', key 'password' value '123456', key 'listenPort' value '6971', key 'isOnline' value 'true', key 'friends' value (empty), key 'email' value 'test1@hotmail.com', key 'connectPort' value '6974', key 'cachedMess...' value (empty), and key 'Address' value '::ffff:127.0.0.1'.

图 1.41 Test1 登录后服务端界面

密码找回测试

首先使用户 Test1 退出登录。然后假设 Test1 用户忘记密码，点击登录界面上的“forget password?”按钮，并在弹出的对话框中填入自己的邮箱，待服务器返回问题 abc 后填入答案 123，然后服务器验证通过，此时两个密码输入框解锁，在其中输入两遍修改后的密码 abcdef，如图 1.42 所示，点击 submit 按钮后对话框关闭，密码修改成功。此时在服务端点击用户 Test1 可以看到其密码已经被更改为 abcdef，如图 1.43 所示。

当用户端输入的用户邮箱未注册过，或者输入的安全问题的答案错误，或者两遍密码输入不一致时，状态显示区会显示相应的错误，并回退到密码输入的初始状态，提示用户重新进行输入。

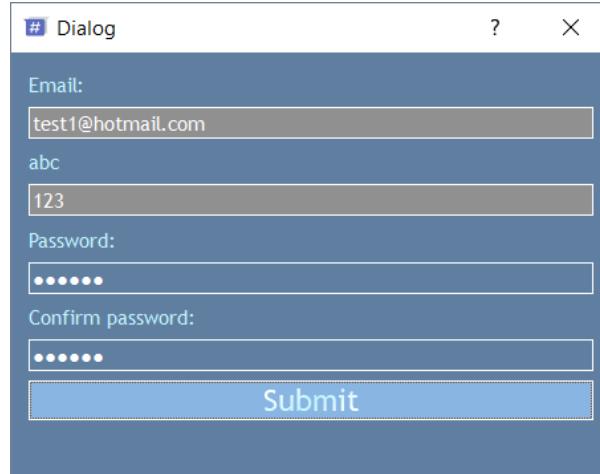


图 1.42 修改用户密码

MainWindow		
validate password	Search	
change passwd	Database	
	email	username ip@connectPort@listenPort
	test4@hotmail.com	Test4 offline
	test3@hotmail.com	Test3 offline
	test2@hotmail.com	Test2 offline
	test1@hotmail.com	Test1 offline

key	value
userName	Test1
safeQuestion	abc
safeAnswer	123
password	abcdef
listenPort	6971
isOnline	false
friends	
email	test1@hotmail.com
connectPort	6974
cachedMess...	
Address	::ffff:127.0.0.1

图 1.43 密码修改成功后的服务端界面

多用户同时在线聊天测试

同时令 Test1、Test2、Test3、Test4 四个用户登录，然后让其两两之间互相发送消息，用以测试是否所有消息都能够成功被接收。其中，Test1 的消息记录如图 1.44 所示。其它客户段的消息记录都与之类似此处不再给出。可以看出，用户之间发送的消息都能够被成功接收，多用户同时在线聊天测试通过。

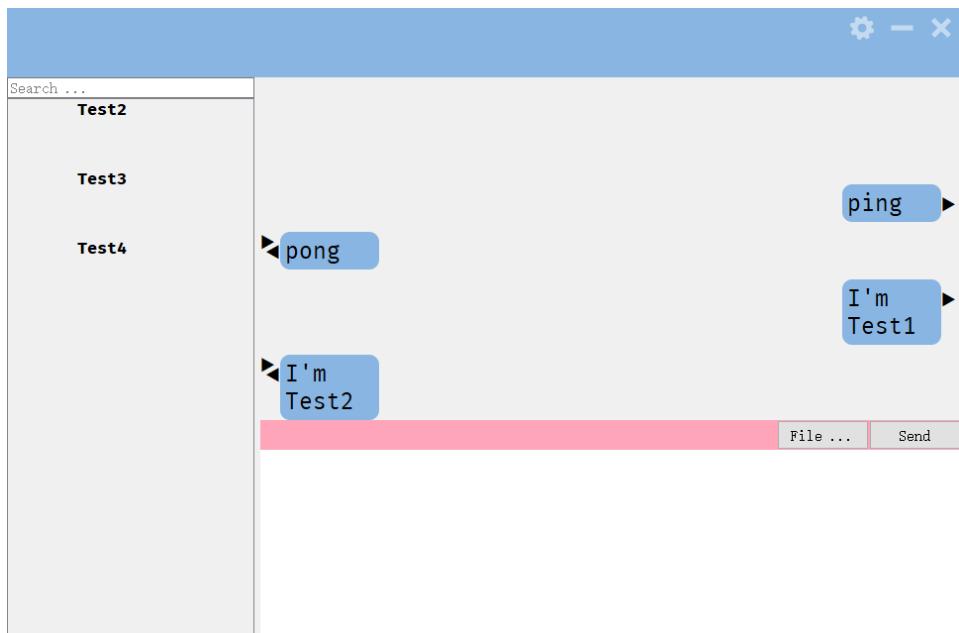


图 1.44 Test1 的消息记录

离线消息测试

对于离线消息的测试，首先让用户 Test1 登出，然后让在线的用户 Test2 给用户 Test1 发送两条消息“Offline message”和“Offline message2”，最后使用用户 Test1 登录，观察 Test1 是否能够成功接收到消息。用户 Test2 给 Test1 发送消息后，服务端界面如图 1.45 所示。可以看出，Test1 用户的 cachedMessage 中出现了 Test2 的用户邮箱以及对应的两条消息，说明消息已经缓存在服务器上。然后使用 Test1 的账号登录，登录后点击 Test2，如 所示，离线的消息被取回，此时服务器中 Test1 中的 cachedMessage 也被清空，说明离线消息功能正常。

MainWindow			key		value
validate password	Search		userName	Test1	
change passwd	Database		safeQuestion	abc	
	email	username	ip@connectPort@listenPort		
	test4@hotmail.com	Test4	offline		
	test3@hotmail.com	Test3	offline		
	test2@hotmail.com	Test2	::ffff:127.0.0.1@1630@1615		
	test1@hotmail.com	Test1	offline		
			cachedMessages		
			test2@hotmail.com	Offline message	
			test2@hotmail.com	Offline message2	
			Address		

图 1.45 离线消息发送后的服务端界面

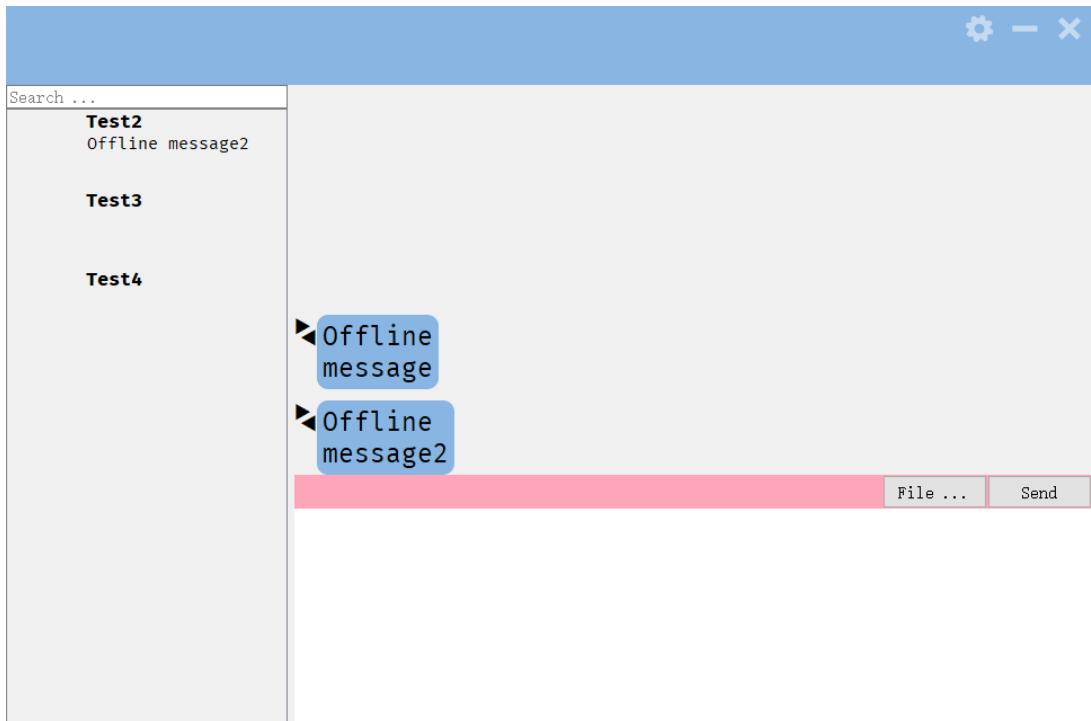


图 1.46 用户 Test1 登录后界面

文件传输测试

令用户 Test1 和 Test2 同时处于在线状态，并让 Test1 给 Test2 发送一个较小的文件，观察 Test2 是否能够正常接收。当 Test1 发送文件后，Test2 显示如图 1.47 所示，文件栏中出现了 Test1 发送的文件并询问是否接收，当 Test2 点击 Accept 后文件开始接收，其速率实时显示如图 1.48 所示。接收完毕后，文件名下方将显示文件接收的平均速率。至此文件传输测试完成，功能正常。

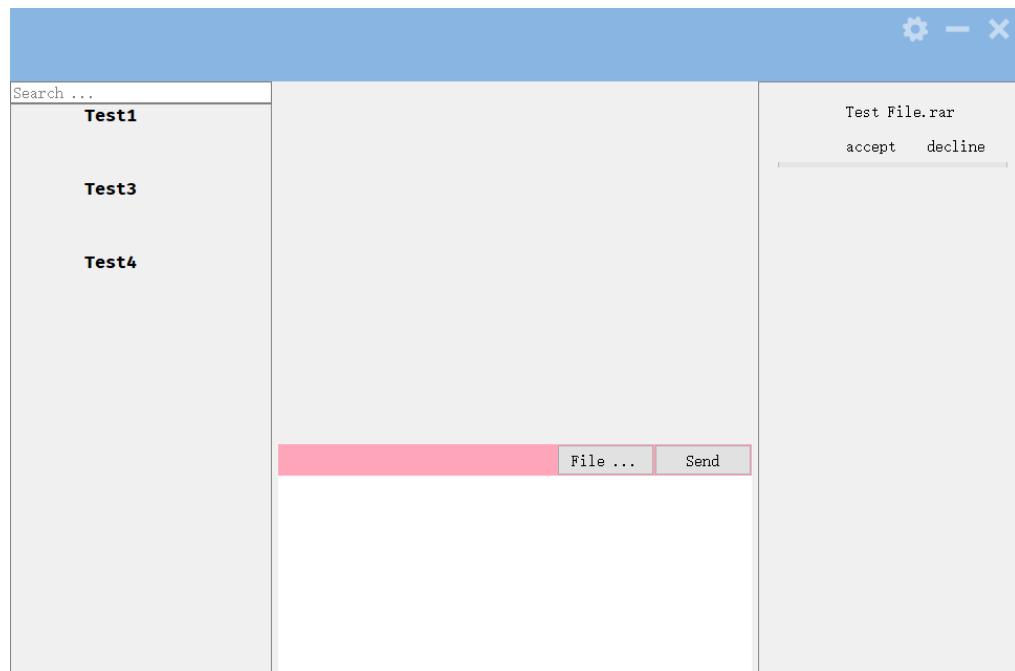


图 1.47 文件已发送但未被接收

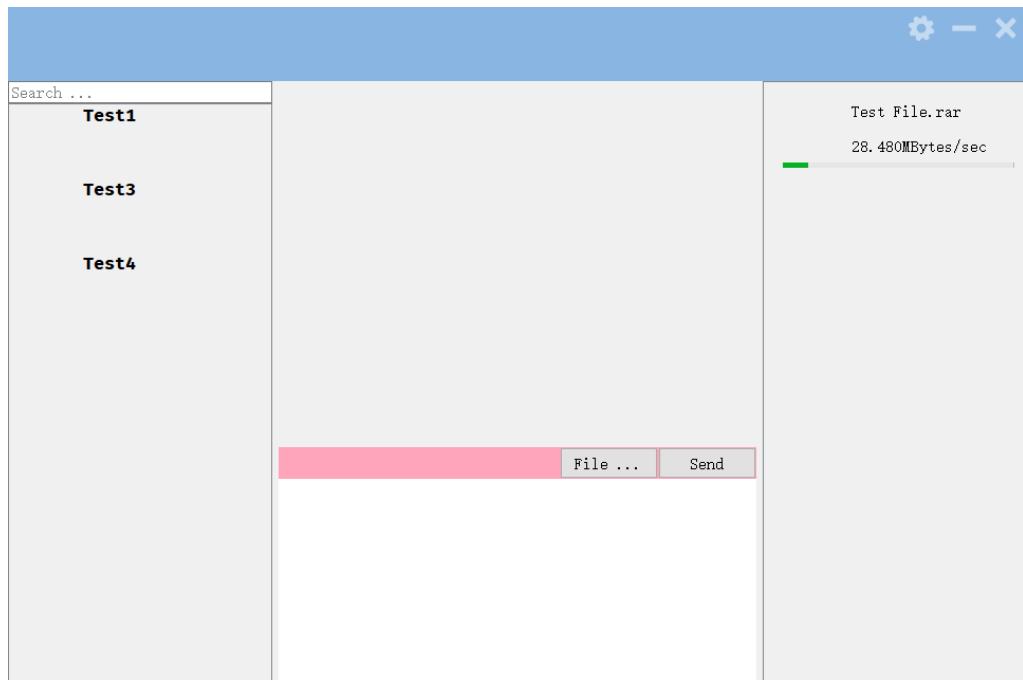


图 1.48 文件接收中

数据库加解密测试

首先关闭服务端程序令其进行保存,然后在 MSYS2 下使用 cat 和 hexdump 工具查看保存的文件,如图 1.49 所示。可以看出,读出的为已经经过加密的二进制内容。再次打开服务端程序并验证密码,如图 1.50 所示,服务端又能读出明文内容,说明加密解密功能正常。

```

fish /home/HuSixu/Project/LowRC/build-LowRCServer/Desktop_Qt_static_MinGW_w64_64bit_MSYS2-Release/release/datas/test1@hotmail.com
~/P/L/b/r/d/test1@hotmail.com
> cat _info
p
> l j h xb . F C9 W        4 eBFKu tFD
EW }s Oq 'bC1 # Q |> I Q pN |UNI v =~ K Wk 9 R- 2
i n
~/P/L/b/r/d/test1@hotmail.com
> hexdump -C _info
00000000  70 0c 58 67 37 5a 1f a3  a8 d4 bc 80 51 8c 9b 7e  |p.Xg7Z.....Q..~|
00000010  00 da fa 44 94 3c a2 0d  05 d4 3e 8b 6c 19 9e 6a  |...D.<....>.l..j|
00000020  a7 d2 16 86 d8 68 b6 78  62 dd e3 2e 9e c8 a8 7f  |.....h.xb.....|
00000030  86 03 08 c7 f7 d8 1f c1  b3 f9 9e 46 d3 43 39 10 |.....F.C9.|
00000040  d4 57 16 09 34 f4 8d f7  e3 17 d3 99 42 96 08 46 |.W..4.....B..F|
00000050  1e 4b 75 9d de cf 74 46  44 0c fc 80 a8 98 9b bb |.Ku...tFD.....|
00000060  0e 10 41 d2 ed 16 88 6e  7f d8 08 60 5e fe 79 4e |..A....n..:^.yN|
00000070  f3 11 14 07 f1 d2 74 57  3b 6a c1 32 a4 a4 e6 45 |.....tW;j.2...E|
00000080  1f 57 a7 81 7d 73 a9 f0  bc 4f 71 90 e5 ba 27 62 |.W..}s...0q...'b|
00000090  43 31 f6 f0 f1 09 ad 23  eb 1e 51 7c 3e 9b c6 c6 |C1.....#..Q|>...|
000000a0  b6 49 b5 93 94 51 87 b7  a0 d2 e8 fe a2 70 4e dd |.I....Q.....pN.|
000000b0  7c c8 96 4e 49 c7 e4 76  e6 ab 3d 3c 7e 1d 81 f8 |..NI..v..=~...|
000000c0  ca 4b 09 c9 57 6b a4 39  9f ea 88 52 2d f9 99 bf |.K..Wk.9...R-...|
000000d0  a3 ce 32 8a 10 1c a0 10  c5 69 c1 ea 06 6e c9 d7 |..2.....i....n..|
000000e0

```

图 1.49 Test1 的用户信息

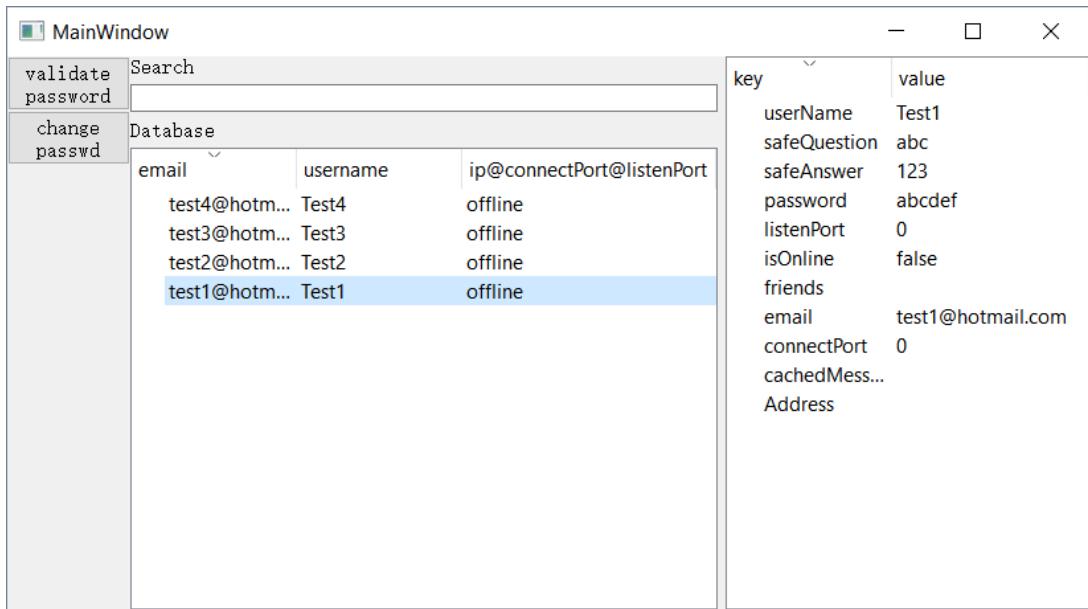


图 1.50 再次打开服务端程序解密

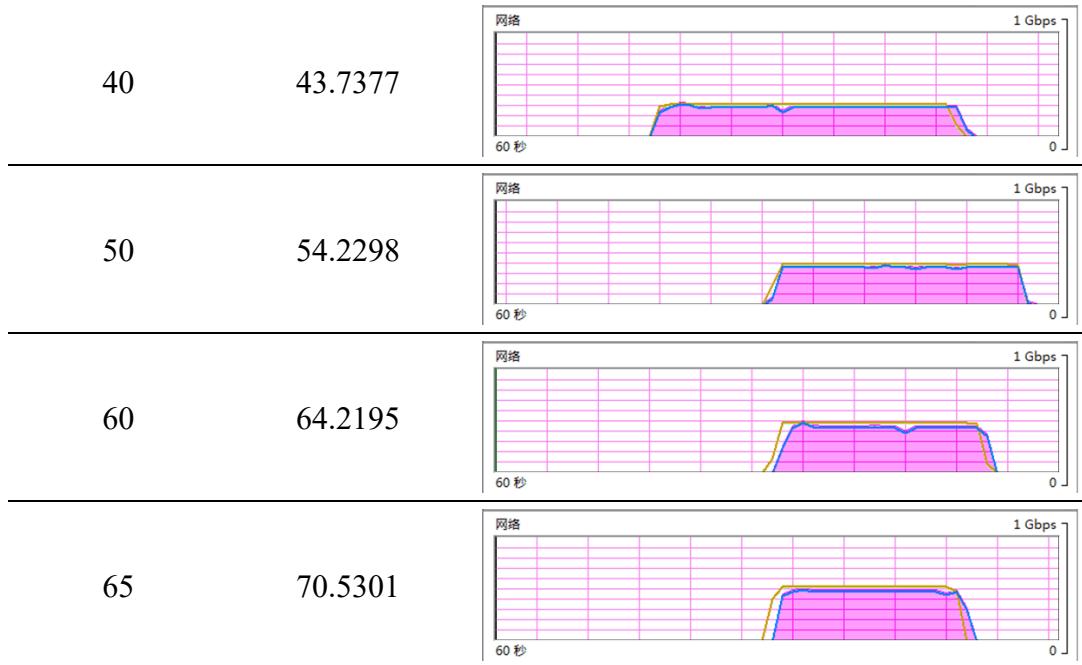
1.5.3 性能测试

性能测试中，由于处于同一局域网中且几乎没有拥塞，发送方发送消息后接收方立即能收到，其延迟肉眼不可测，已经达到要求的水平。又由于本协议的设计中离线消息的获取是随着登录信息一同返回的，因此在测试中，在登录成功后立即能看到所有的离线消息，不存在延迟。

对于文件传输速率的测试中，传输一大小为 1.3GB 的文件，在客户端配置使用不同的 UDP 包大小进行测试，并通过资源监视器观察测试结果，其结果如表 1-3 所示。由于系统的限制，单个 UDP 包的大小不能超过 65536 字节，在实际测试中若包的大小超过 65536 字节程序将会报错，因此测试用的最大包大小为 65000 字节。又由于 Qt 定时器最高精度为 1ms，因此将定时器设置为 1ms。此时的理论速度上限为 $65000\text{Bytes} \div 1\text{msec} = 65\text{MBytes}$ ，由于 Qt 定时器并不是精准定时 1ms，因此最后测得的最高速度为 70.53MBytes/s。

表 1-3 不同包大小下的传输速率及速度曲线

单个包大小 (KBytes)	平均速率 (MBytes/sec)	速度曲线
20	21.8227	
30	32.8003	



在测试时，曾尝试不间断发送数据，但发现无论包的大小是何值（取 100 字节、500 字节、1K 字节、5K 字节、10K 字节进行测试），丢包率均很高，传输速率均未能超过 2MBytes/s，推测是由于处理器速度过快造成的网络压力超过负载，因此在其后的测试中加入了 1ms 的发送间隔，此后的测试中即使 UDP 包大小为 65Kbytes 丢包率也可以忽略不计。最终测出的平均速度与包大小的关系曲线如图 1.51 所示。从图中可以看出传输的最终平均速率与报告呈明显线性关系。

这与协议设计以及实现所预想的结果一致：在不存在停等，网络状况良好的情况下，数据包的发送间隔不变的情况下，数据发送速率应该与数据包大小呈线性关系。但是由于平台的限制，在测试平台的条件下数据发送速率最大只能达到 70MBytes/s，而理论上限是 125MBytes/s。当尝试使用每一个间隔发送两个数据包时，在报文大小为 50Kbytes 以下时丢包率均约为 50%，而报文大小超过 50Kbytes 时丢包率则会进一步上升，测得的最高传输速率为 54.4179MBytes/s，推测是因为 UDP 的 SendTo 不会造成线程阻塞，也就是说，UDP 的 SentTo 不会像 TCP 中的 SendTo 那样，直到数据完全发送才会 return 回调用函数，它不能保证当执行下一条语句时数据是否被发送。这样，如果要发送的数据过多或者过大，那么在缓冲区满的那个瞬间要发送的报文就很有可能被丢失。在定时器精度不能改善的情况下，进一步提高发送速率应该可以通过多线程发送的方式

进行解决，然而由于这需要对程序进行较大的修改，这一功能未能实现。

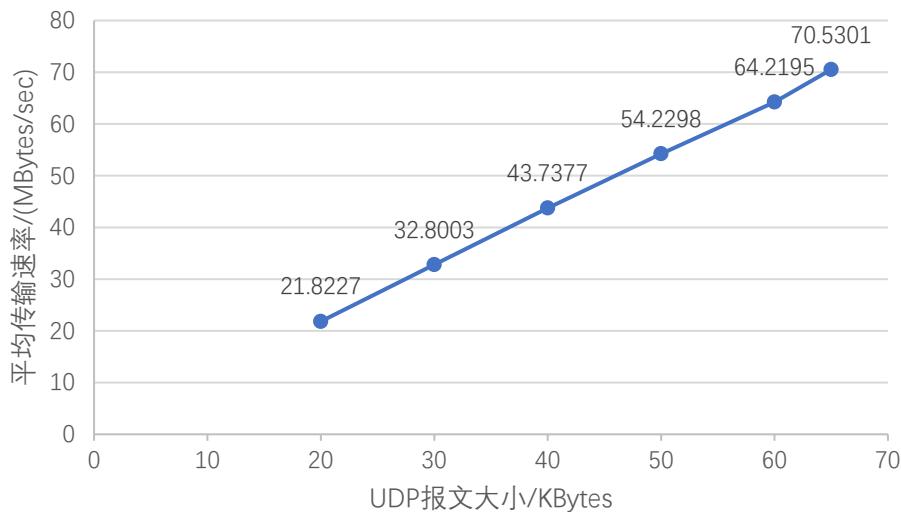


图 1.51 文件传输平均速度——报文大小关系曲线

1.6 其它需要说明的问题

编译指南

本程序开发环境较为特殊，如需进行编译，需要采取以下步骤：

1. 确保使用的机器为 64 位机，在 <http://www.msys2.org/> 下载 MSYS2 64 位版并安装
2. 进入 MSYS2 Mingw64 shell（而不是 MSYS2 shell）
3. 使用 pacman -Syu 命令，并按照提示更新所有源
4. 使用 pacman 包管理工具安装以下软件包或软件包组，安装软件包所使用的命令为 pacman -S <name>，<name>包名或组名。
 - a. base-devel
 - b. mingw-w64-x86_64-toolchain
 - c. cmake
 - d. mingw-w64-x86_64-qt5-static（注意此处必须使用静态版本的 Qt）
5. 接下来可以在 IDE 中进行编译，也可以在命令行中进行编译。在 IDE 中进行编译时配置较为繁琐，此处采取较为简单的直接使用命令编译的方式。此处 bash 下运行 export PATH=/mingw64/qt5-static/bin:\$PATH 添加环境变量，注意一定将 qt5-static 置于开头以确保运行的为静态版本的 Qt。
6. 首先进入 LowRC/LowRCProtocol 中，运行 qmake && make release
7. 查看 LowRCProtocol.a 所处的位置。由于 qmake 会根据平台的不同来决定目标文件夹的位置，因此在模块间依赖时需要手工解决位置问题。

8. 更改 LowRCClient 以及 LowRCSERVER 中的 pro 文件，将 LIB+=后面的路径更改为 LowRCProtocol.a 所在的位置

9. 分别在 LowRC/LowRCClient/以及 LowRC/LowRCSERVER 下运行 qmake && make release 得到两个可执行文件

通过以上步骤可以得到两个可执行文件 LowRCClient 和 LowRCSERVER，分别为客户端和服务端的可执行文件。

其它说明

1. 由于时间不够充裕，很多功能没有实现，报告中对于一些函数也未能进行详细的描述，如需查看文档，在确保下载了 Graphviz 软件并且 MSYS2 Mingw64 的 PATH 环境变量（不是 windows 的环境变量）中存在 dot.exe 可执行文件，并且下载了 mingw-w64-x86_64-doxygen 软件包后，可在 LowRCClient、LowRCSERVER 以及 LowRCProtocol 目录下分别运行 doxygen Doxyfile 命令行编译文档，生成的文档会存在 doc 文件夹下。

2. 在进行课堂检查时最终的速度约为 40MBytes/s 而不是 70MBytes 是由于采取了较为保守的策略以避免发生意外。

3. 所有的代码使用 UNIX 换行符 (LF)，请勿用 windows 记事本打开，也不要将 IDE 的换行显示设置为 Windows 类型 (CRLF)，否则将看到所有的代码都在同一行。

实验二 基于 NS2 的协议分析实验

2.1 环境

处理器: Intel® Core™ i7-6700HQ CPU @ 2.60GHz 2.59GHz
内存大小: 8.00 GB (7.89GB 可用)
操作系统: Archlinux 64 位, 已于实验前对系统进行滚动更新
第三方软件: NSG 2.1 (<https://sites.google.com/site/pengjungwu/nsg>)
Ns2 2.35 patch 版(<https://aur.archlinux.org/packages/ns/>)
Nam 1.15 (<https://aur.archlinux.org/packages/nam/>)

2.2 实验要求

第一项实验——仿真与测试 TCP 和 UDP 协议

- ✧ 网络性能的比较
- ✧ 公平性研究与探讨

第二项实验——仿真与测试 TCP 协议中的不同拥塞控制算法（端到端拥塞控制）

- ✧ TCP Tahoe 算法、TCP Reno 算法、TCP New Reno 算法、TCP SACK 算法、TCP FACK 算法和 TCP Vegas 算法
- ✧ 性能对比
- ✧ 拥塞窗口、阈值变化、吞吐量、网络效率、带宽利用率
- ✧ 拥塞控制能力对比

第三项实验——仿真与测试不同 IP 拥塞控制策略（中间节点排队策略）

- ✧ 先进先出 FIFO、随机早期检测算法 RED、显示拥塞指示算法 ECN、公平排队算法 FQ、随机公平排队算法 SFQ、加权公平排队算法 WFQ
 - 性能对比
 - 阈值变化、吞吐量、网络效率、带宽利用率
 - 拥塞控制能力对比

2.3 实验步骤说明及结果分析

2.3.1 第一项实验的步骤及结果分析

首先在 NSG 中搭建网络，搭建完成后的拓扑结构如图 2.1 所示，其中所有的参数均使用默认参数，生成 Tcl 脚本，然后使用 NS2 进行模拟。

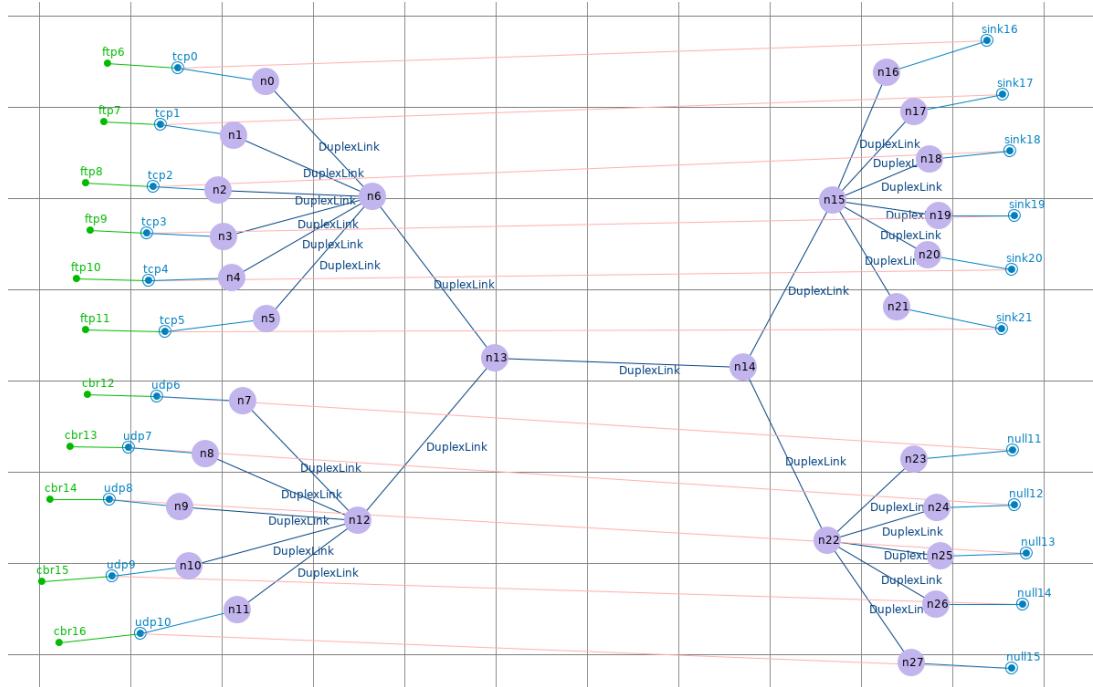


图 2.1 实验一网络拓扑结构

模拟完成后读取并分析生成的文件以进行统计，统计的结果如表 2-1 所示。从表中可以初步看出，各个 TCP 节点间的重传率都比较相近，端到端时延也几乎一样，但使用 TCP 与 UDP 进行对比时发现 UDP 的重传率远低于 TCP 报文的重传率。

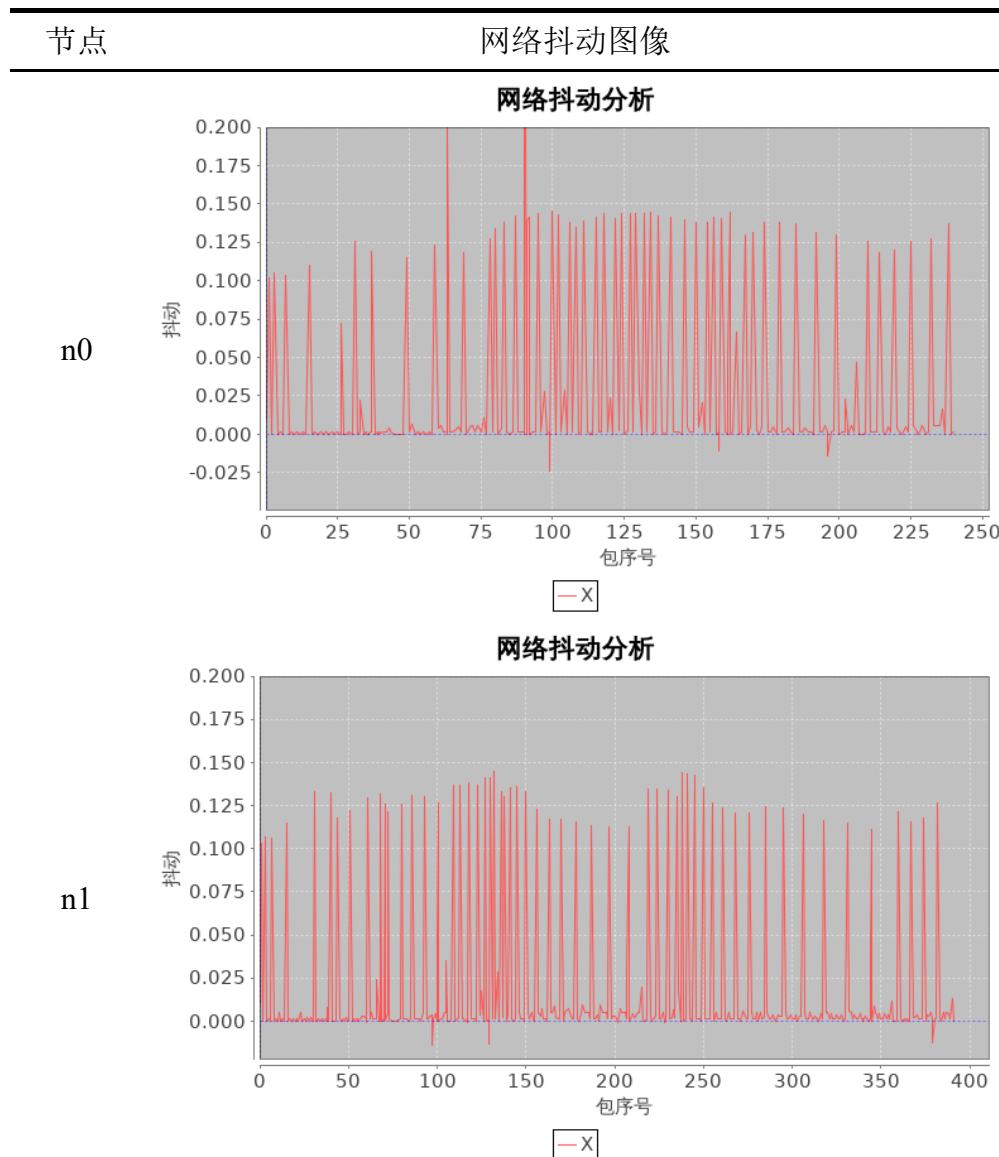
表 2-1 模拟结果统计

报文类型	节点	重传率	端到端时延/s
tcp	n0 → n16	0.07089552	0.101725
	n1 → n17	0.040669855	0.102925
	n2 → n18	0.06993007	0.104125
	n3 → n19	0.071969695	0.105325
	n4 → n10	0.049046323	0.105325
	n5 → n11	0.055194806	0.107725
udp	n7 → n23	0.0025201612	--
	n8 → n24	0.00907258	--

报文类型	节点	重传率	端到端时延/s
	n9 → n25	0.025214322	--
	n10 → n26	0.070095815	--
	n11 → n27	0.13615733	--

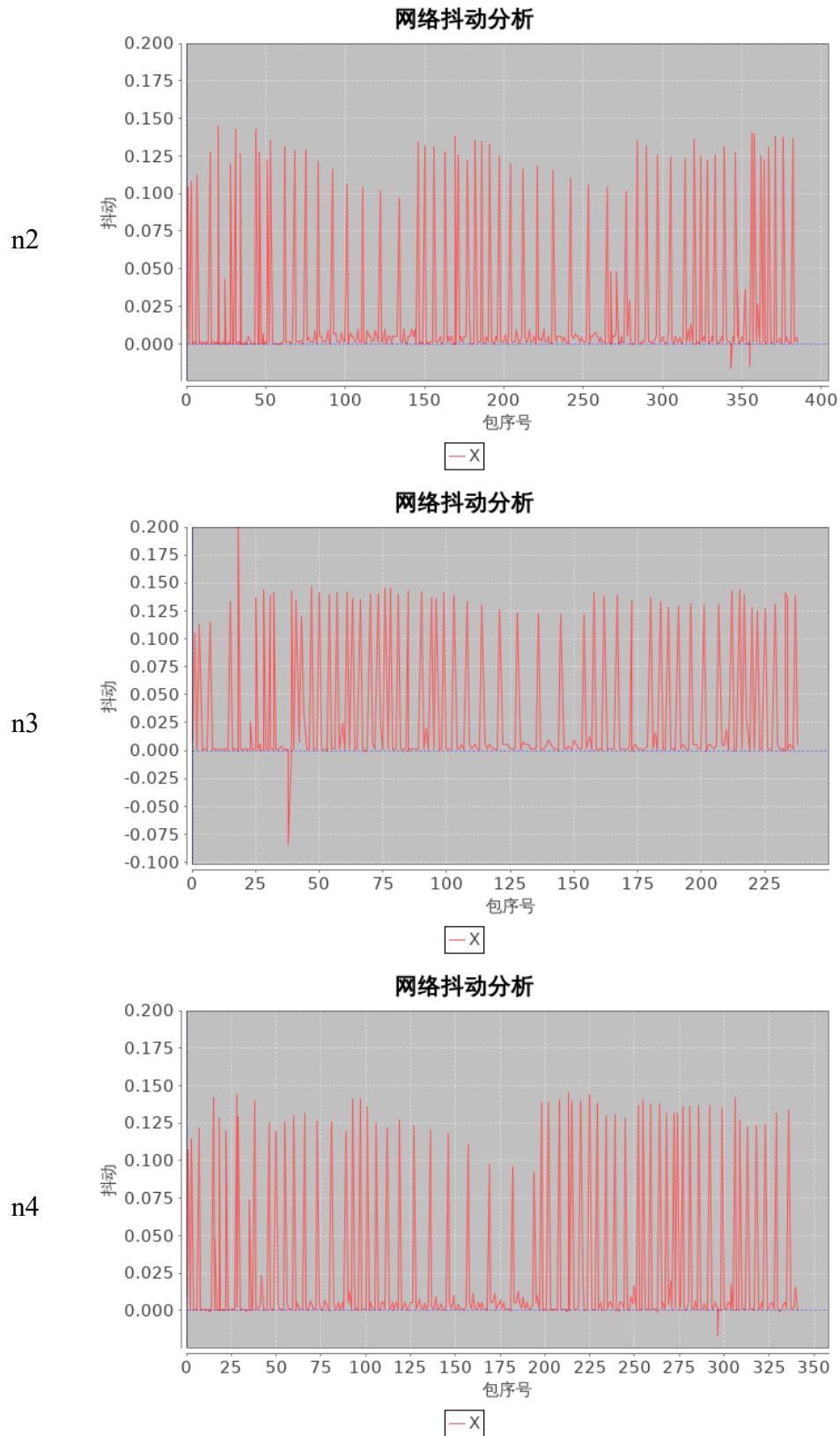
在模拟结束后采集到的发送方各个 TCP 节点的网络抖动图像如表 2-2 所示。网络抖动描述的是分组端到端时延的变化。在本实验的结果中，6 个 TCP 节点的网络抖动图像类似，其频率均在 4~10 个包一次，峰值平均在 0.125 左右，且鲜有负值出现，说明在当前状况下，发送方平均网络时延在逐渐上升。

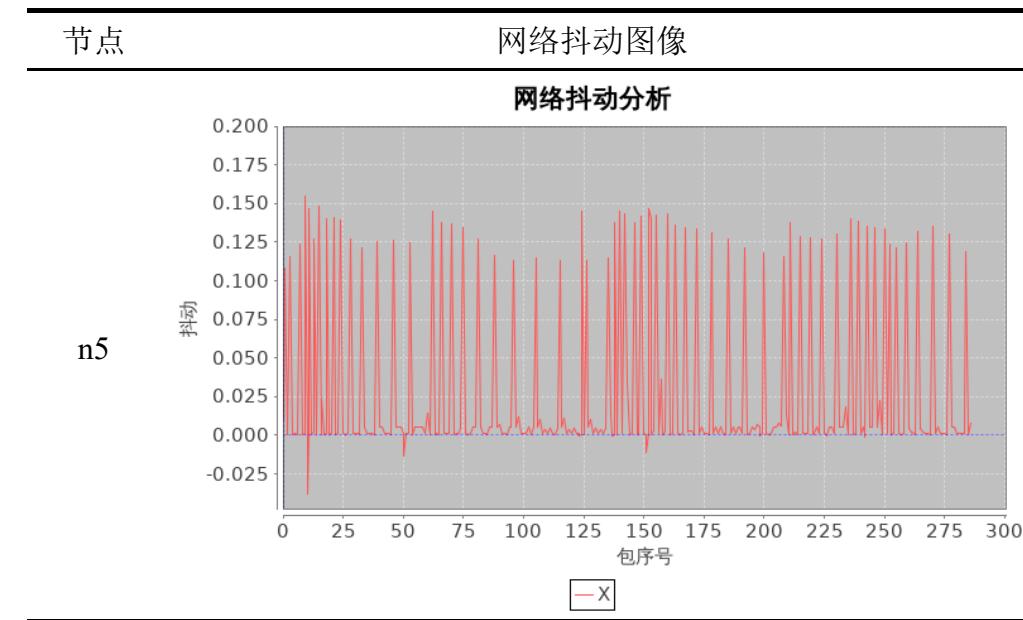
表 2-2 各发送节点网络抖动图像



节点

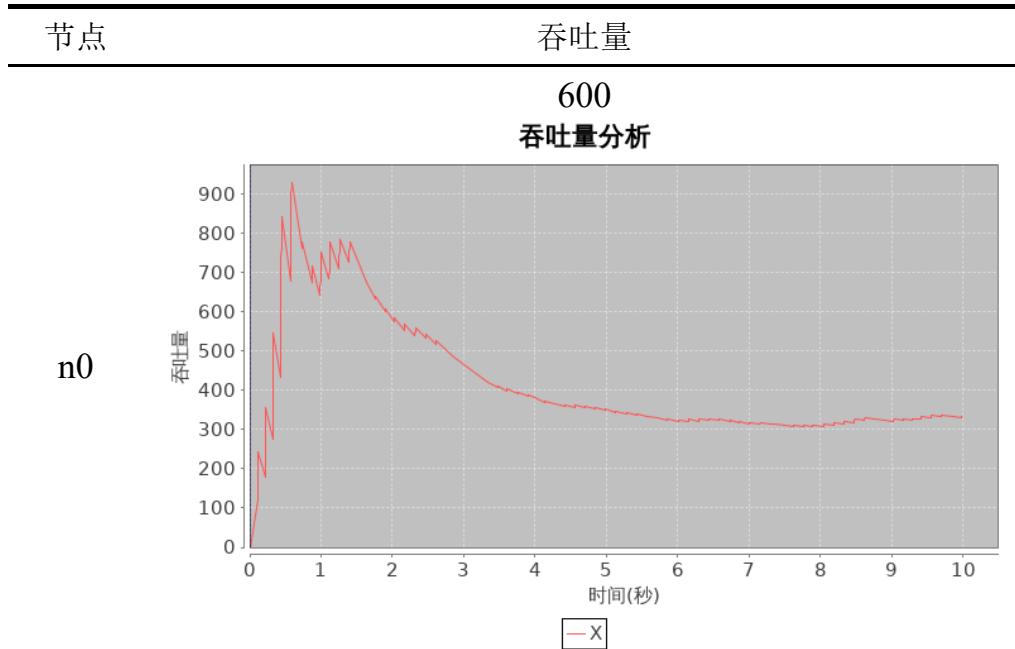
网络抖动图像





在最终的结果中、各个 TCP 节点随时间的网络吞吐量图像如表 2-3 所示。网络吞吐量为单位时间内网络上能够成功传输的平均数据量，即实际的网络速率。可以看出各个发送节点推图量的明显变化规律：首先在很短的时间内上升到一个较大值，然后较快的下降并逐渐稳定在一个稳定的值上。各个发送节点的吞吐量峰值和稳定值差异较大，但基本在同一个数量级上，说明默认的拥塞控制协议还是基本做到了在 TCP 间的公平的。但值得注意的是，各个 UDP 发送节点的吞吐量图像均如图 2.2 所示，可以看到，各个 UDP 节点的吞吐量均稳定在 1600 左右，远远超出 TCP 发送节点的稳定位置，说明网络对于 TCP 和 UDP 而言是不公平的，UDP 占据较大的优势。

表 2-3 各发送节点吞吐量随事件变化图

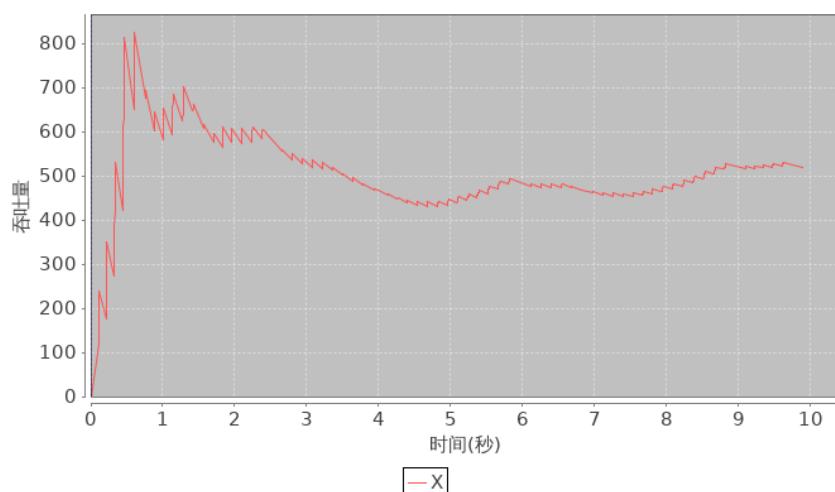


节点

吞吐量

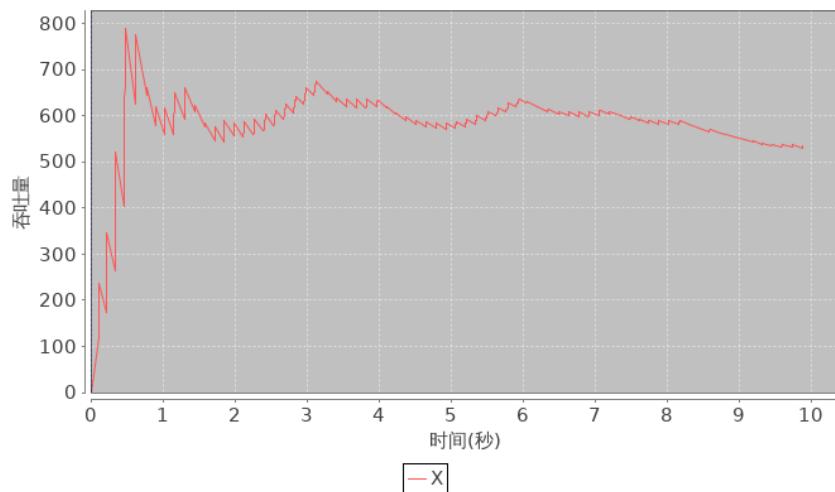
吞吐量分析

n1



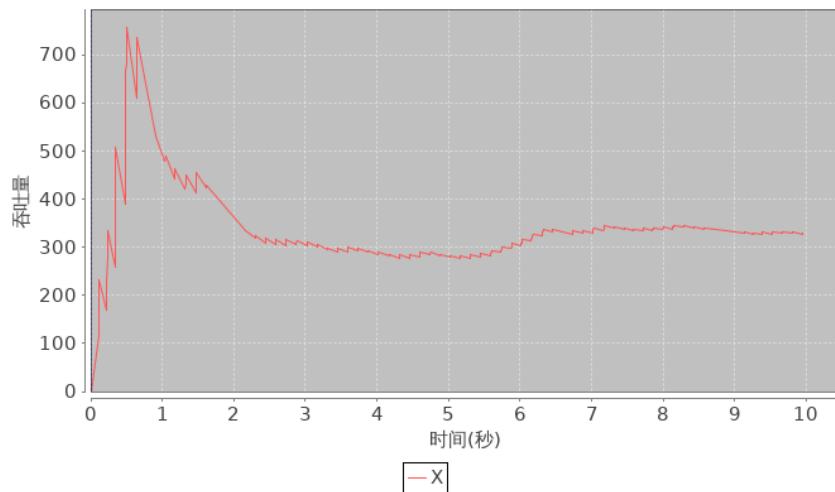
吞吐量分析

n2



吞吐量分析

n3



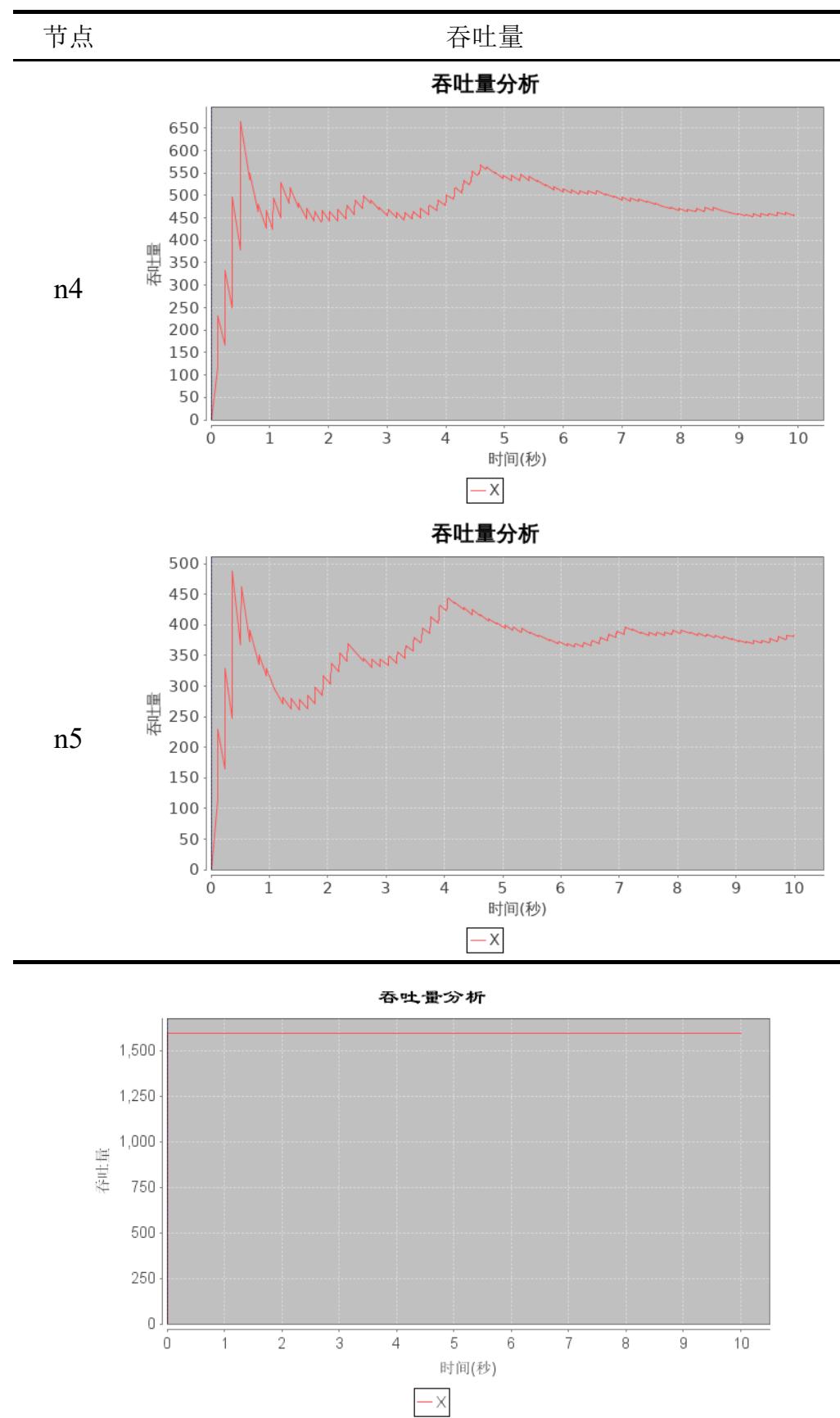
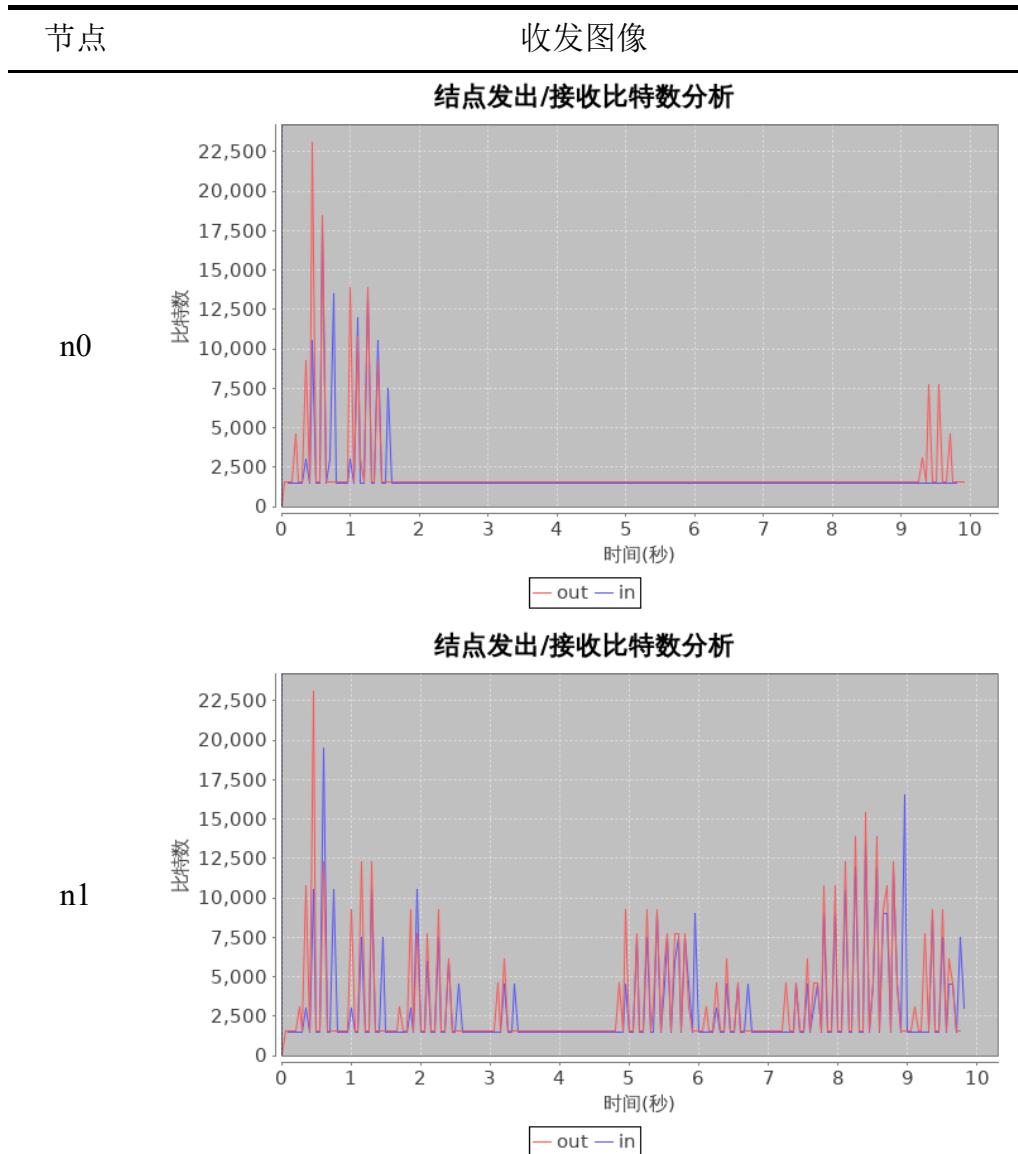


图 2.2 UDP 节点的吞吐量图像

各个 TCP 节点的发送/接收比特数随时间变化的图像如表 2-4 所示。从图中可以看出，各个 TCP 节点有数据包发送的时间都不相同，所有节点均有一段或多段时间稳定在一个较低值，说明在网络负载较大的情况下，默认的 TCP 拥塞控制协议并不能很好的利用网络带宽。

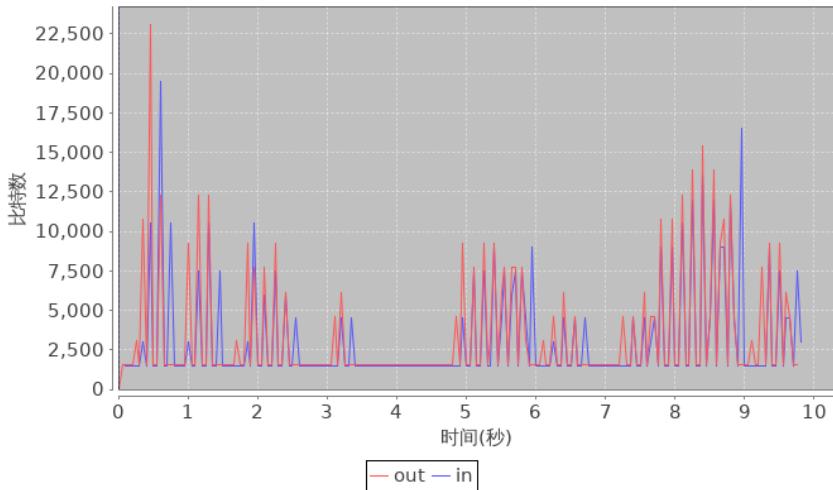
表 2-4 各节点收发比特数随时间变化图像



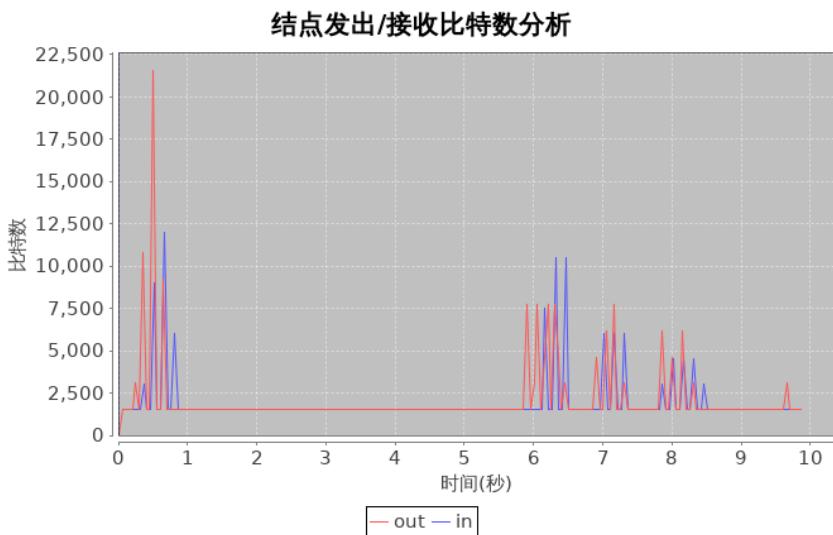
节点

收发图像

结点发出/接收比特数分析



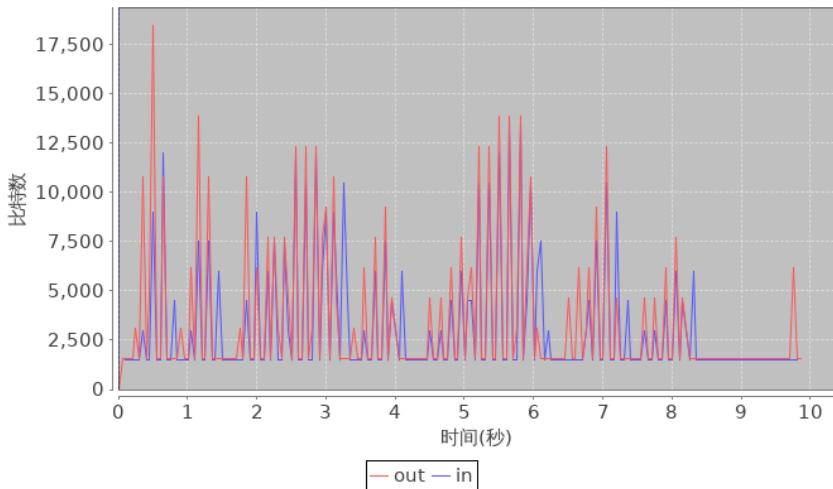
n3



7

结点发出/接收比特数分析

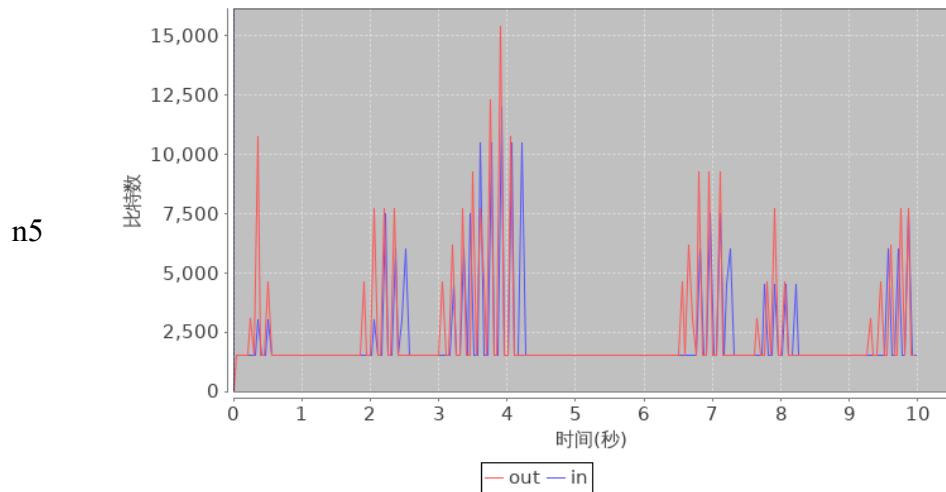
n4



节点

收发图像

结点发出/接收比特数分析



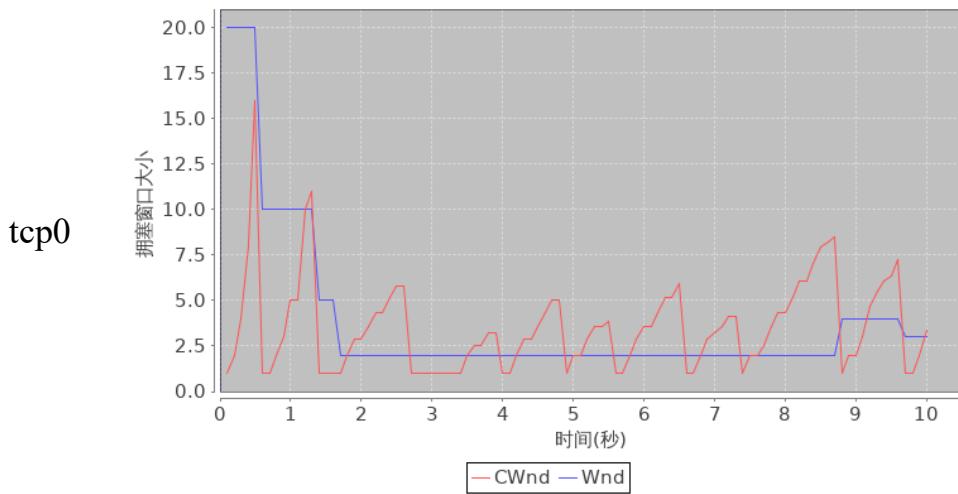
各个 TCP 节点的窗口变化图像如表 2-5 所示, 可以看出, 这些图像有强烈的 TCP Tahoe 拥塞控制算法的特征: 在 Cwnd 值以下, 窗口呈指数扩大, 每次大小翻倍, 而对于超过 Cwnd 的部分, 窗口线性增长, 在发生丢包的情况下, 更新 CWnd 大小, 窗口大小跌回 1 并重新开始增长。从图中可以看出, 由于网络压力较大, 最终 CWnd 都稳定在了非常低的值, 且网络抖动十分剧烈, 这都是 Tahoe 算法需要改进的地方。

表 2-5 各 TCP 代理窗口变化图像

代理

窗口变化图像

拥塞窗口分析

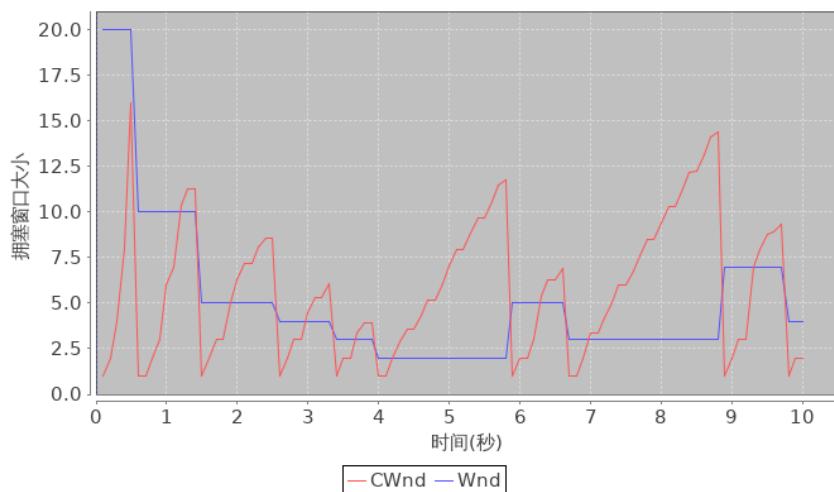


代理

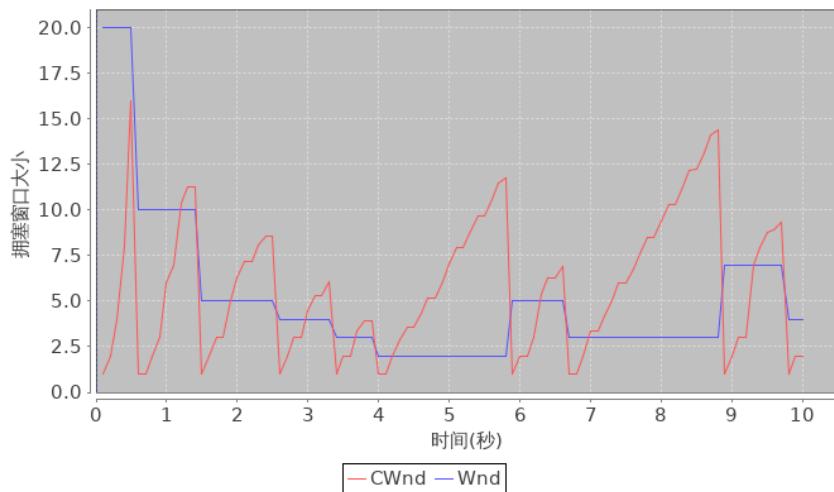
窗口变化图像

拥塞窗口分析

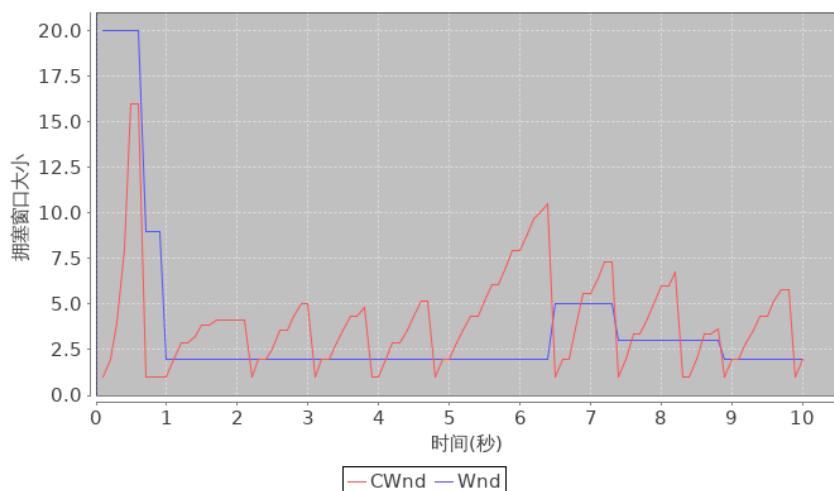
tcp1



tcp2



tcp3

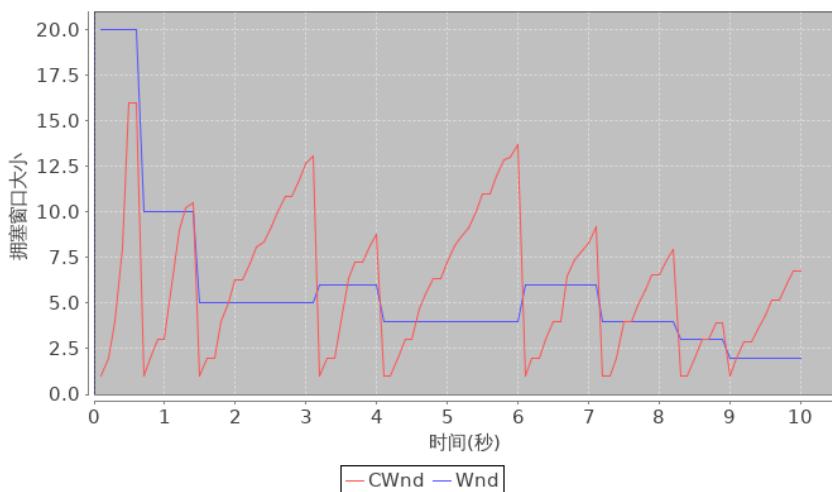


代理

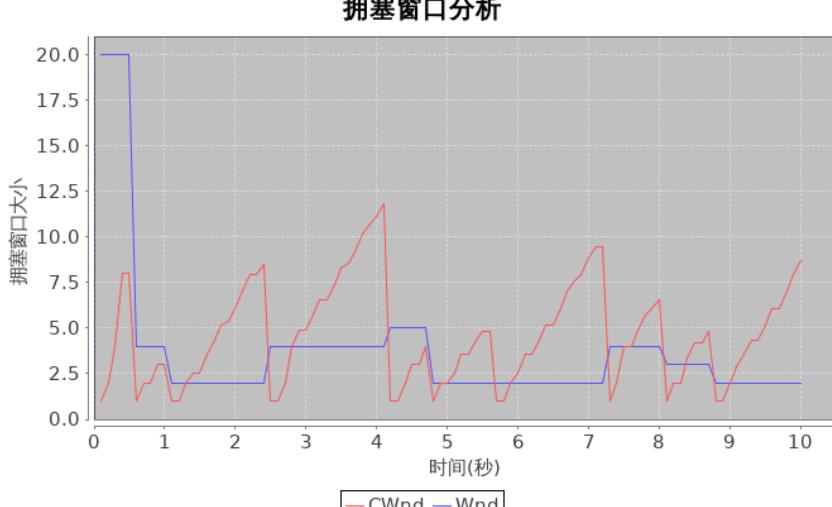
窗口变化图像

拥塞窗口分析

tcp4



tcp5



2.3.2 第二项实验的步骤及结果分析

第二项实验的网络拓扑与第一项相同，但各个 TCP 传输层代理的拥塞控制协议进行了修改。分别更改为 Tahoe、Reno、newReno、Vegas、SACK、FACK 协议，更改后的网络拓扑结构如图 2.3 所示，其节点 n26 到 n27 中心链路的带宽设置为 10Mbps，其余链路带宽均为 100Mbps。同样使用 NSG 生成 Tcl 脚本后使用 NS2 进行模拟，然后分析所得到的 out.tr 跟踪文件获得结果。

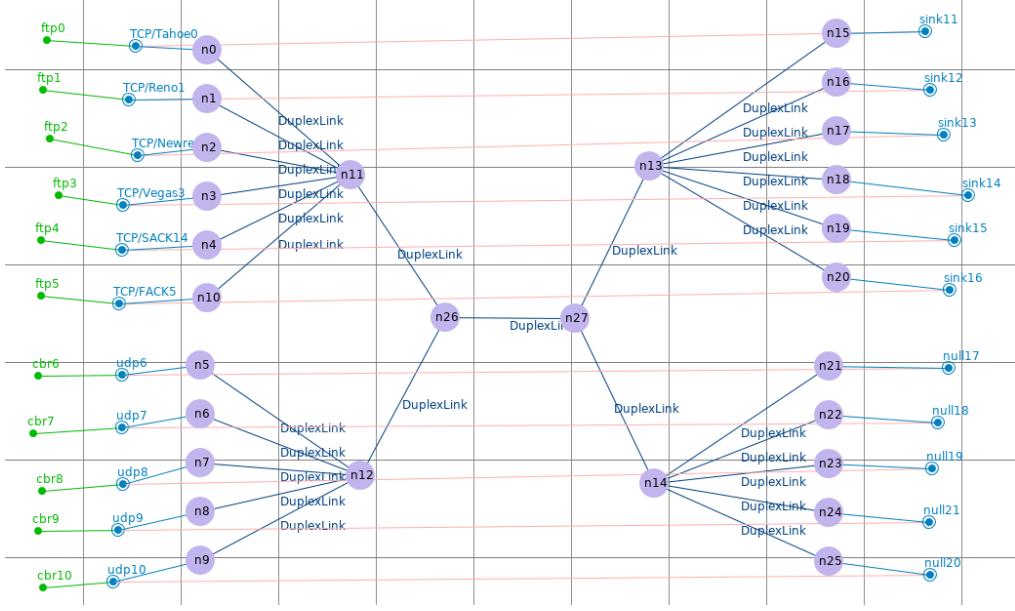


图 2.3 更改 TCP 传输层协议后的网络结构

对于模拟的输出文件，其统计结果如表 2-6 所示。可以看出，在其它条件相同观点情况下，各个 TCP 拥塞控制协议的包重传率不禁相同，气质过来以 Vegas 协议下的包重传率最小，而 Fack 协议下的重传率最大。当所有的控制协议的平均时延相近。

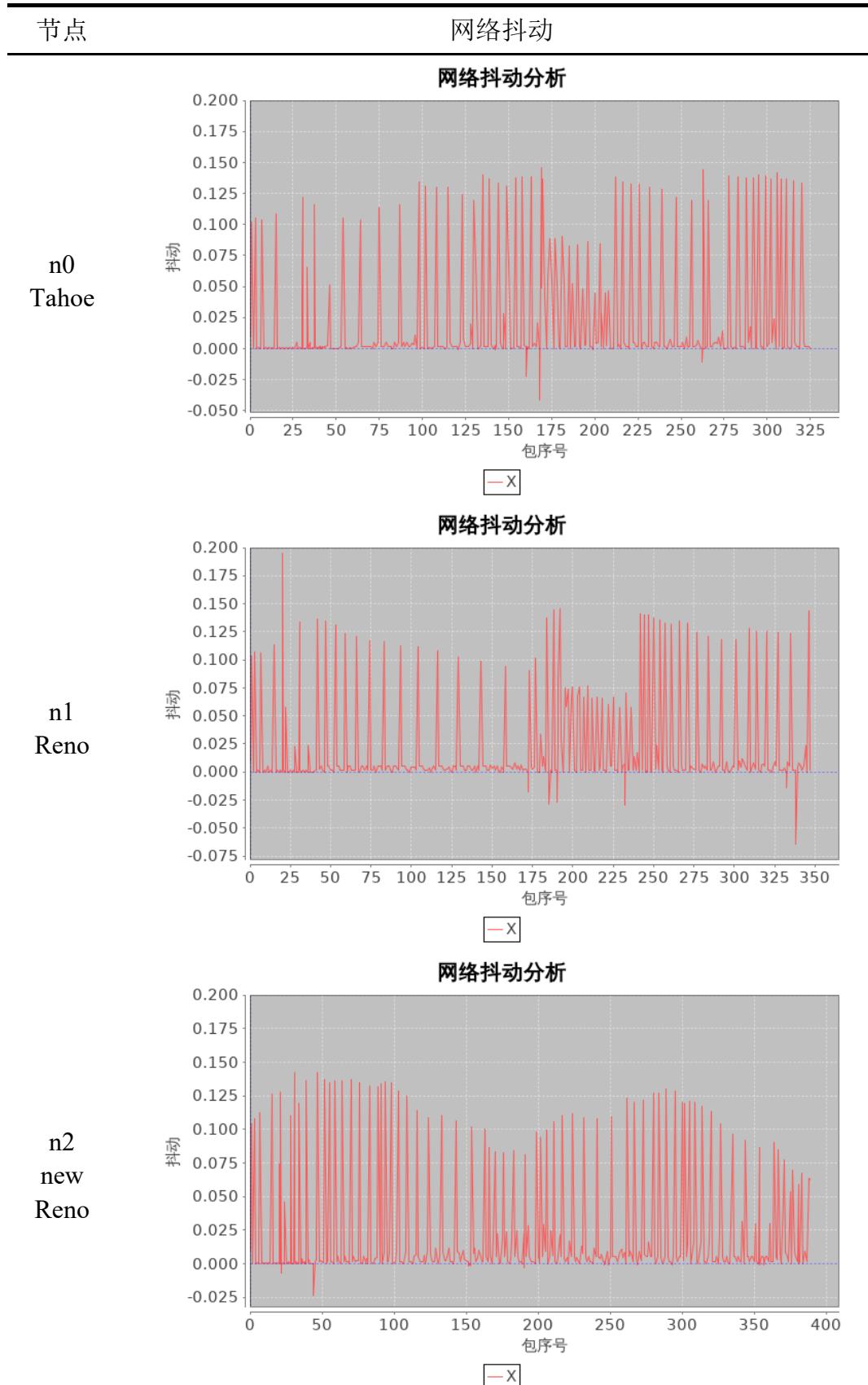
表 2-6 实验 2 模拟结果统计

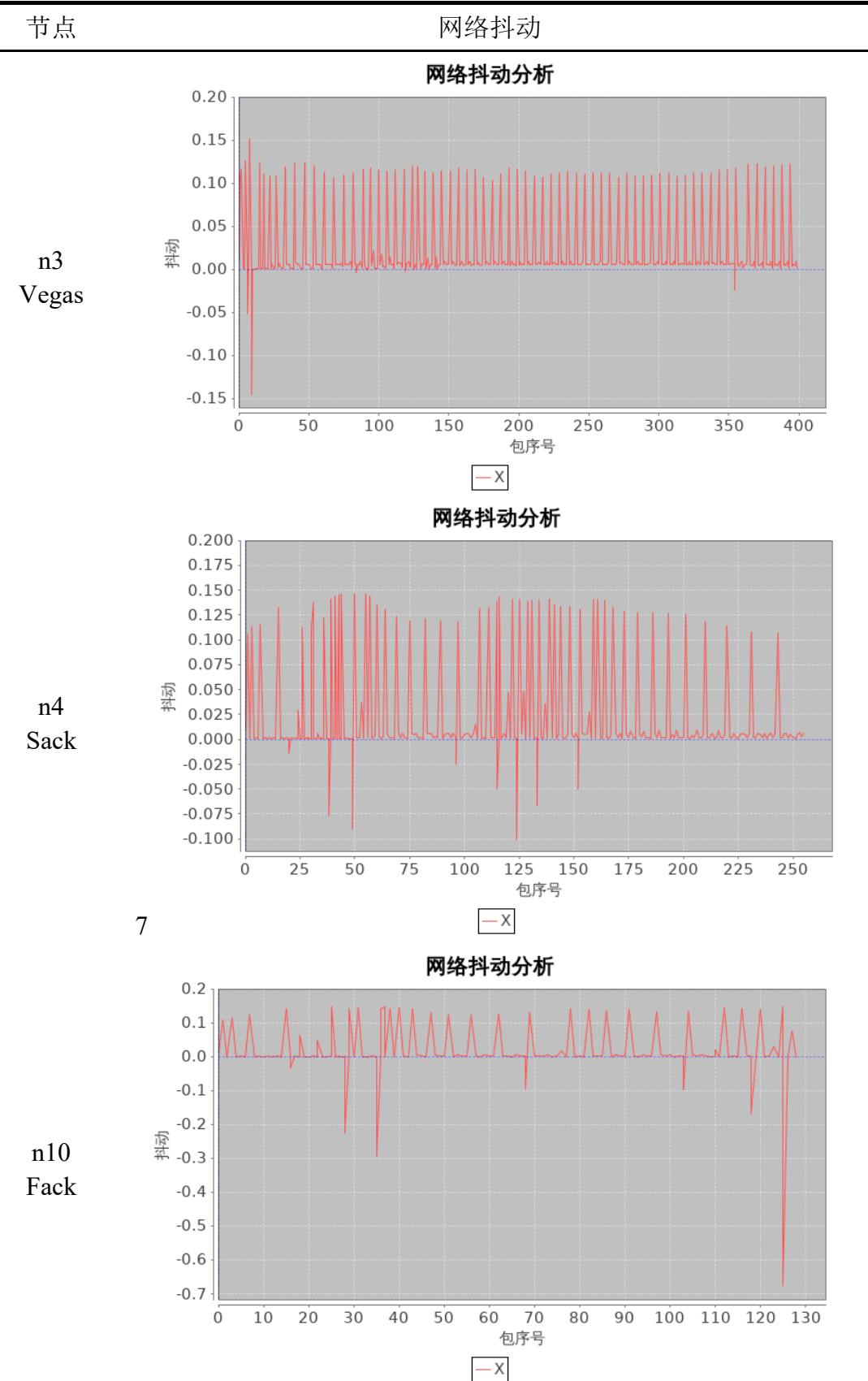
网络类型	链路	重传率	平均时延
tcp/Tahoe	n0 → n15	0.053672317	0.101725
tcp/Reno	n1 → n16	0.040871933	0.102925
tcp/newReno	n2 → n17	0.05437352	0.104125
tcp/Vegas	n3 → n18	0.0125	0.107725
tcp/Sack	n4 → n19	0.07266436	0.105325
tcp/Fack	n10 → n20	0.115384616	0.106525
	n5 → n21	0.0020181634	0.0
	n6 → n22	0.0060544903	0.0
	n7 → n23	0.018677436	0.0
udp	n8 → n24	0.050984353	0.0
	n9 → n25	0.1207851	0.0

类似地，对于跟踪结果中的网络抖动、吞吐量、收发比特数以及窗口变化进行分析。其中，网络抖动的图像如表 2-7 所示。从图中可以看出，各个拥塞控制协议的网络抖动都是不同的。对于 Tahoe、Reno、newReno 和 Sack 而言，其网络抖动幅度较大且不规则，而 Vegas 网络抖动峰值较小且频率稳定，Fack

协议网络抖动最小，且频率较低，说明 Fack 协议对于网络的稳定性的控制强。

表 2-7 网络抖动图像

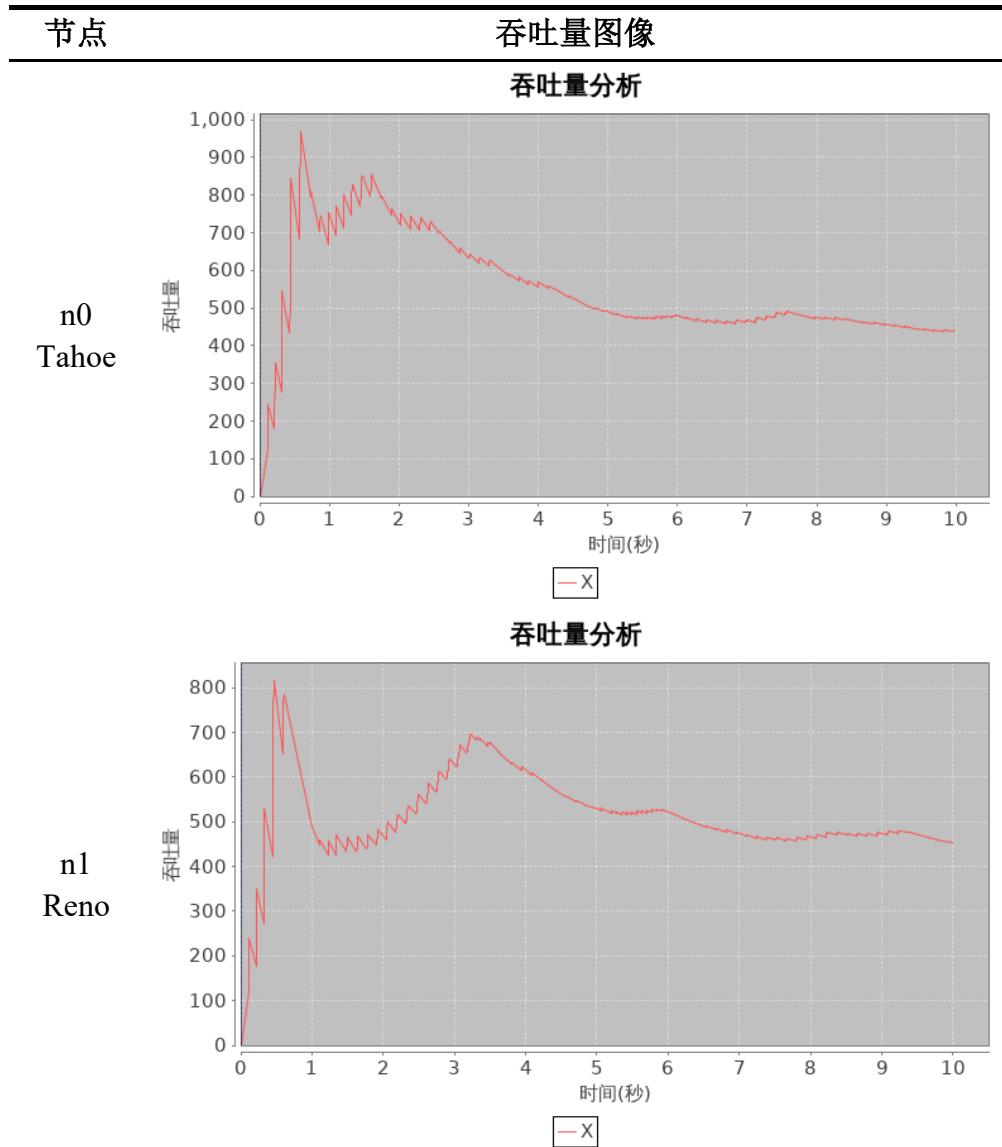




各个使用不同 Tcp 拥塞控制协议节点的吞吐量图像如表 2-8 所示。从图中可以看出，除了 Vegas 协议外，其它协议控制的节点的吞吐量均是先上升到一

个峰值后再下降到一个稳定的值，而 Vegas 则是持续上升直到稳定。这实际上与 Vegas 协议的特性有关，它是基于 TTL 的协议，随着 TTL 的上升而逐渐减小。在最后稳的吞吐量中，Fack 协议的吞吐量最低，这是由于 Fack 在重传过程中作了拥塞流控，导致其吞吐量较低。

表 2-8 各 TCP 节点吞吐量随时间变化图

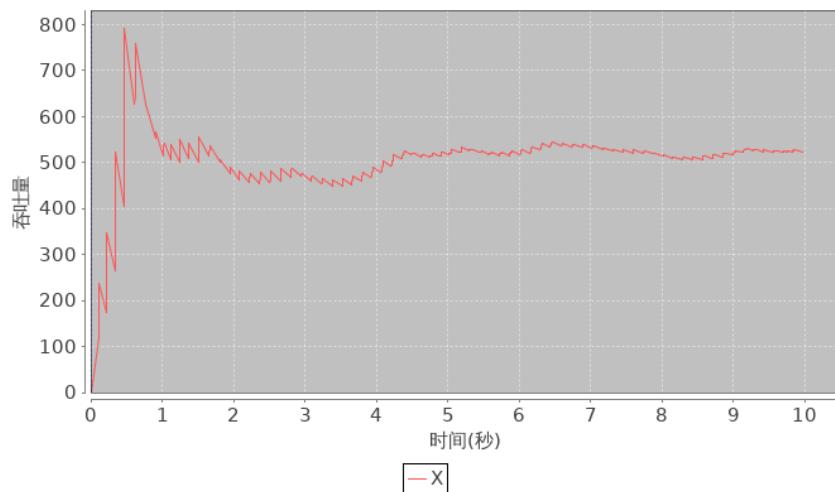


节点

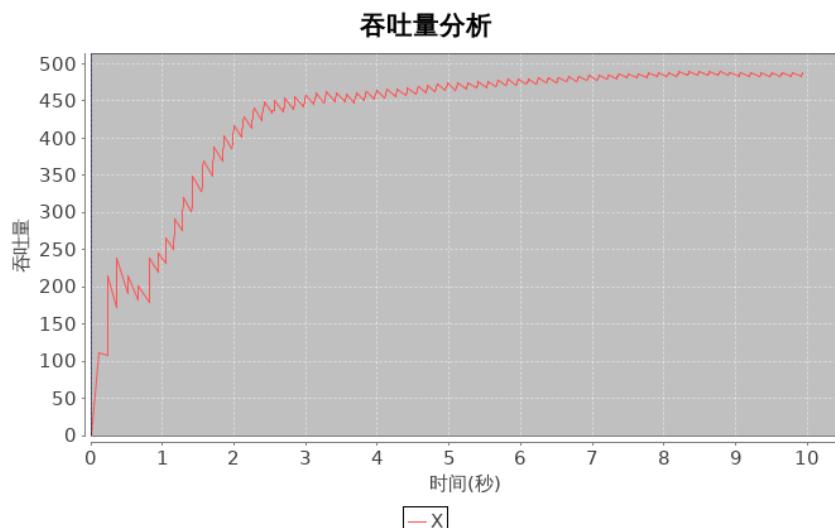
吞吐量图像

吞吐量分析

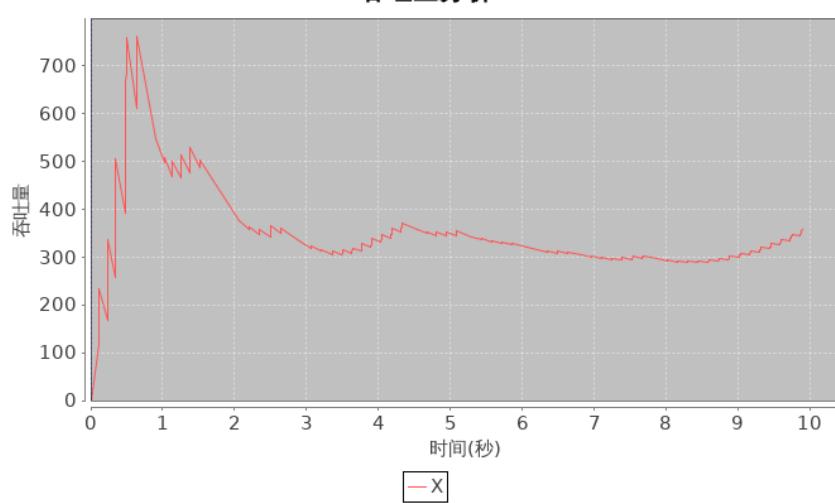
n2
new
Reno



n3
Vegas



n4
Sack

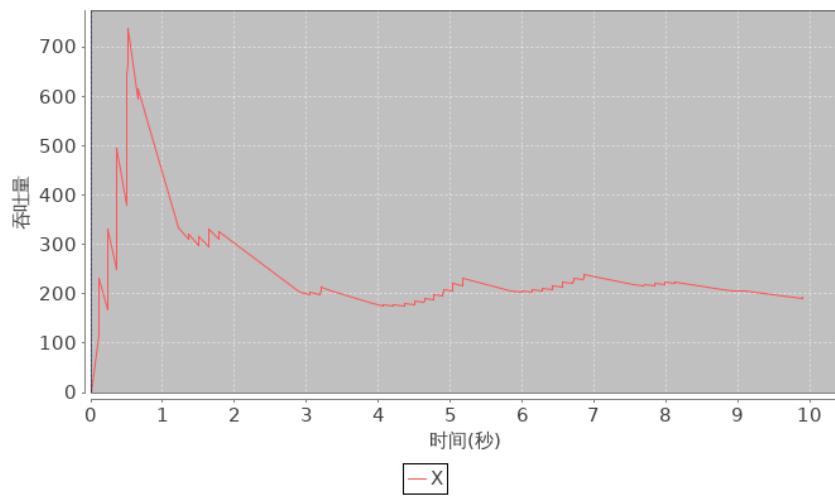


节点

吞吐量图像

吞吐量分析

n10
Fack



各个 TCP 的收发比特数图像如表 2-9 所示，从各图中可以看出，Reno 的发送频率密度大于 Tahoe, newReno 又大于 Reno, 这是由于 Reno 在出现丢包时窗口大小没有减到 1 而是减半, newReno 通过 Partial ACK 机制延长了 Fast Recover 和 Fast Retransmit 的过程。Vegas 时基于 TTL 的协议，因此发送/接收比特数最为稳定。Fack 与 Sack 相比发送比特数明显较低，是由于 Fack 对重传过程中流量更为严格的控制。

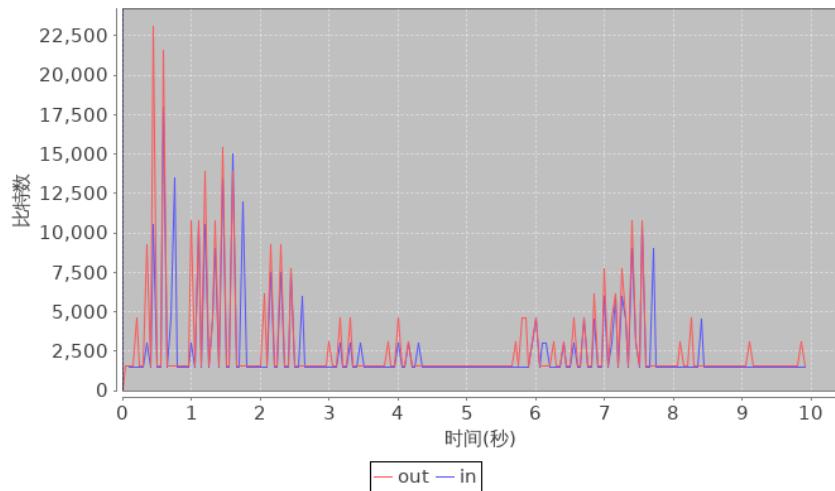
表 2-9 各 TCP 节点收发比特数随时间变化图像

节点

收发比特数图像

结点发出/接收比特数分析

n0
Tahoe

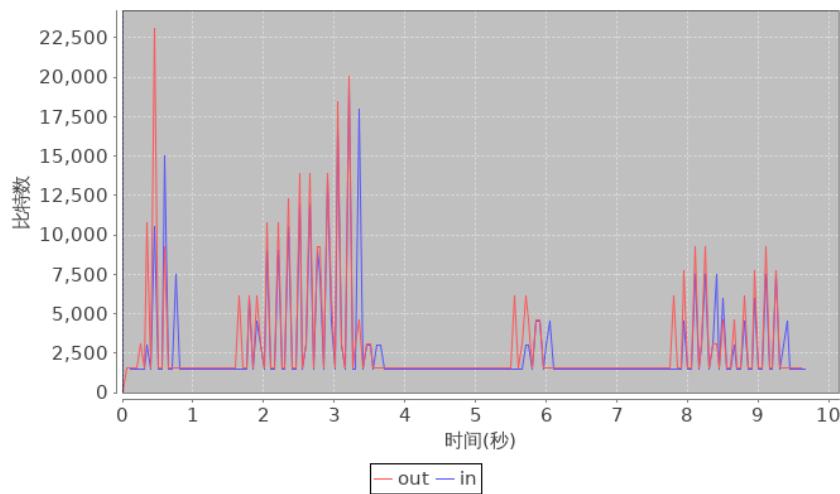


节点

收发比特数图像

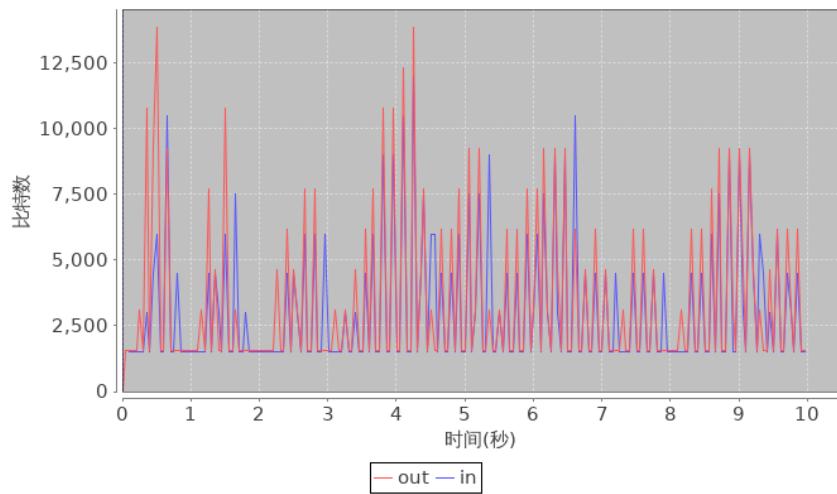
结点发出/接收比特数分析

n1
Reno



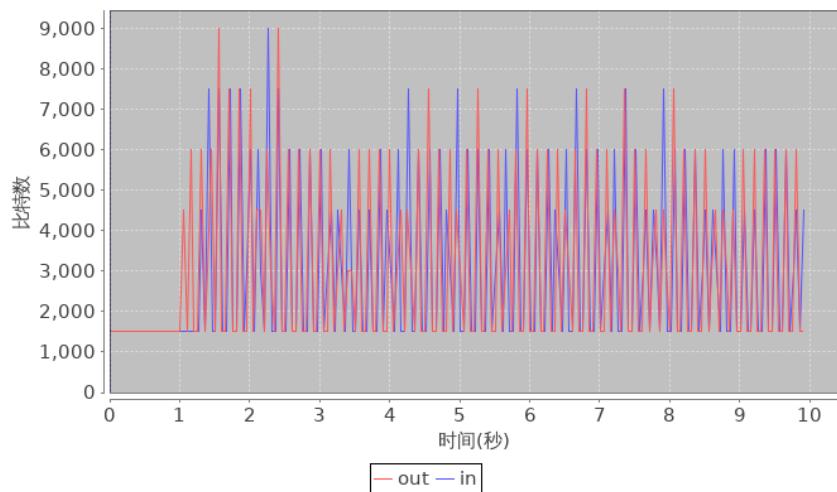
结点发出/接收比特数分析

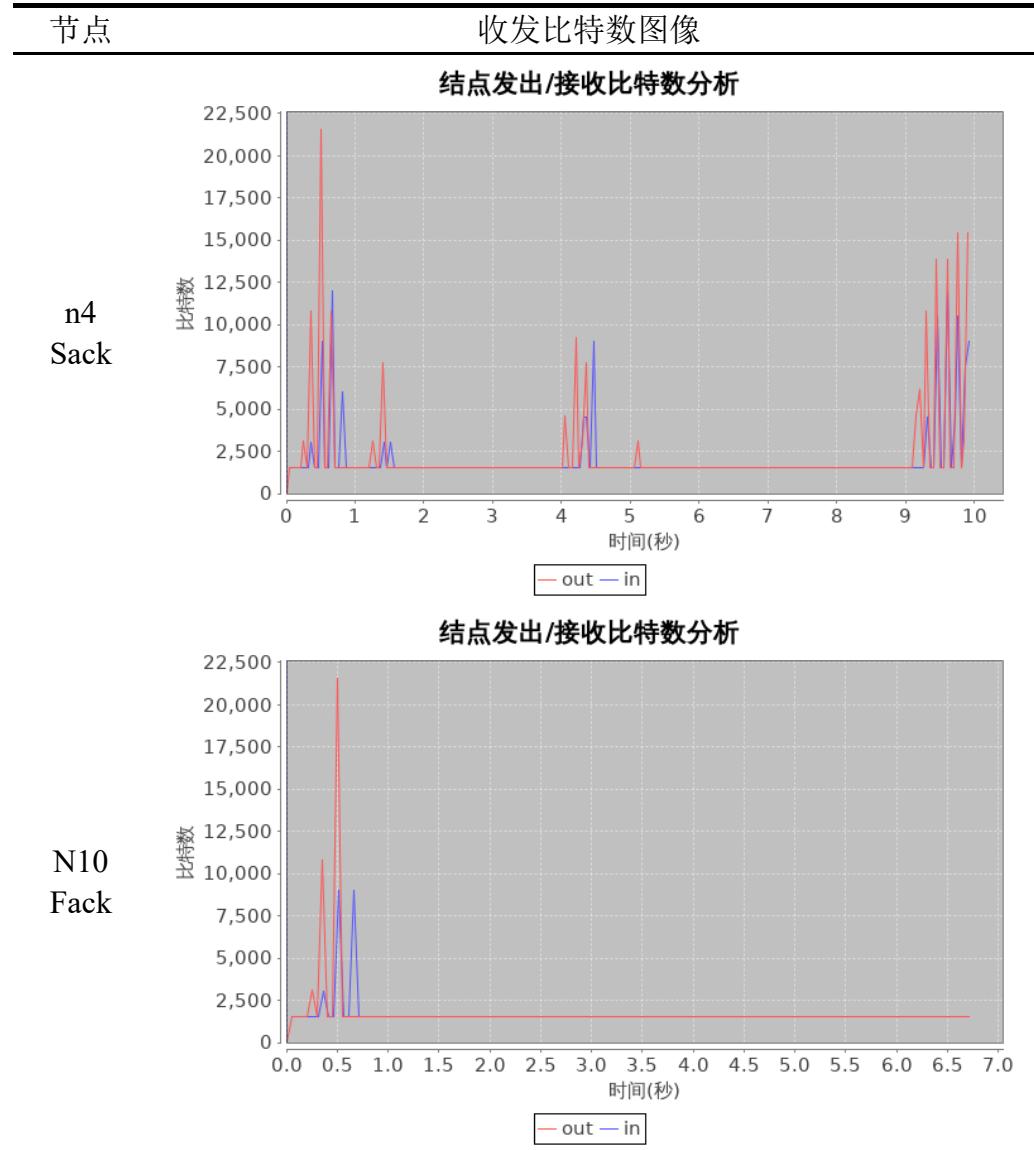
n2
new
Reno



结点发出/接收比特数分析

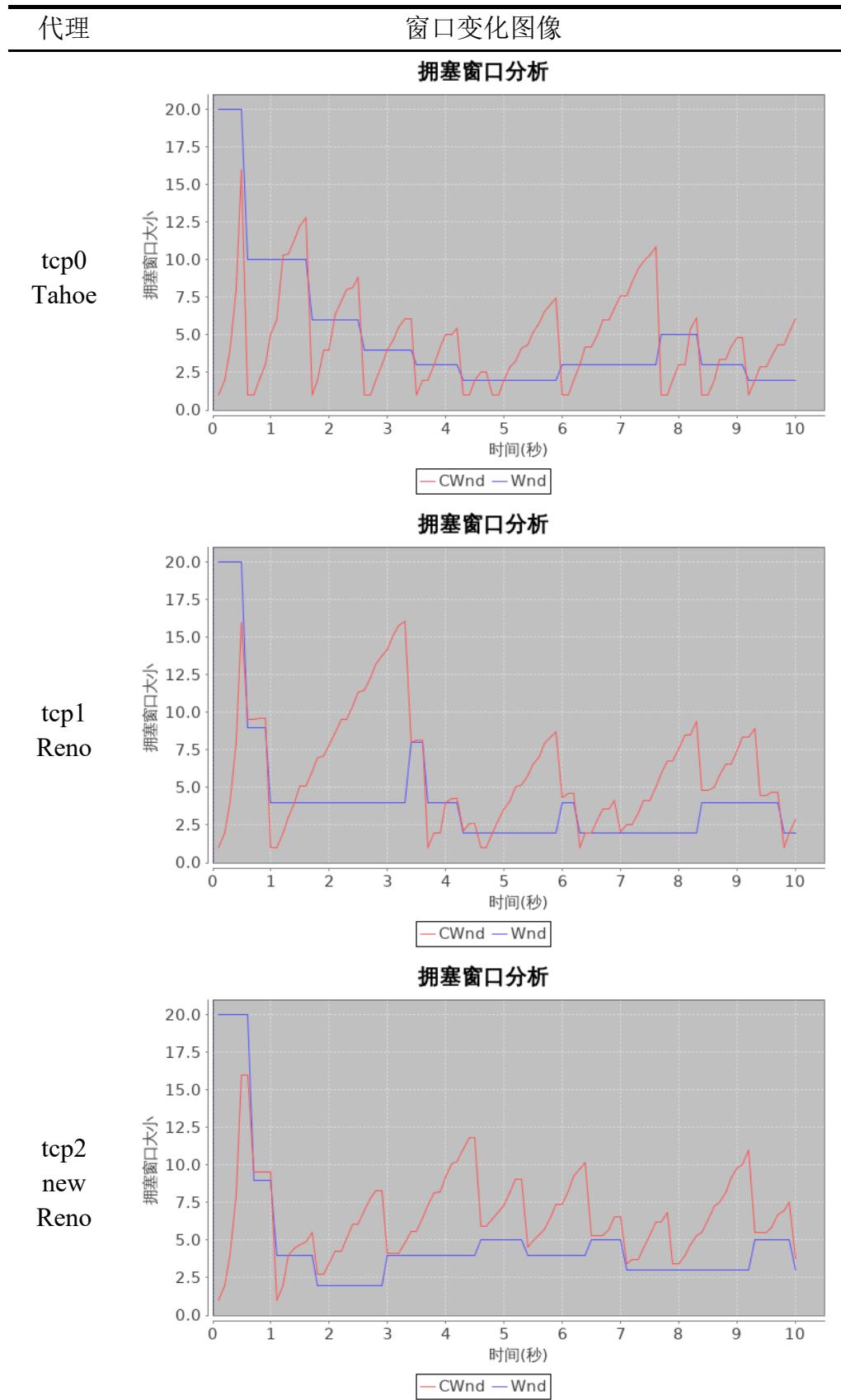
n3
Vegas





各个 TCP 传输层代理的窗口变化图像如表 2-10 所示。从图中可以清晰的看出各个协议的特征。对于 Tahoe 而言，当窗口大小没有超过 CWnd 时，窗口大小呈指数增长，超过后呈线性增长。发生丢包后 CWnd 大小变为窗口大小的一半然后窗口大小直变为 1，与协议内容实现一致。而 Reno 协议中发生丢包时窗口大小变为原来的一半而不是变为 1。newReno 与 Reno 类似，它与 Reno 的改进并不能在这张图上体现。Vegas 协议的曲线则较为特殊，窗口大小很快就稳定在一个值，几乎不发生变化，十分稳定。Sack 和 Fack 则与 Tahoe 类似，只不过其使用的为选择重传而不是全部重传。在同一个网络环境下不同的拥塞控制协议的竞争力不同，其中以 Vegas 最为稳定，此时对于使用不同协议的 Tcp 代理而言，网络环境不是公平的。当然 UDP 对于 TCP 始终占据着优势。

表 2-10 各 TCP 代理窗口变化图像

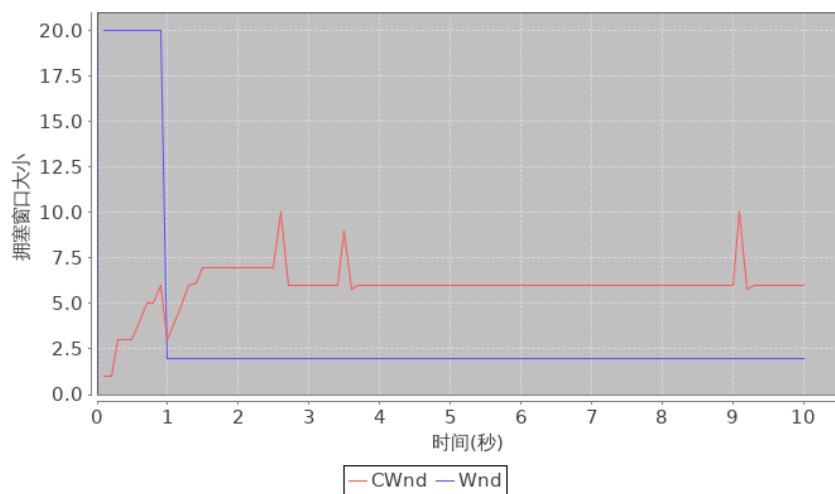


代理

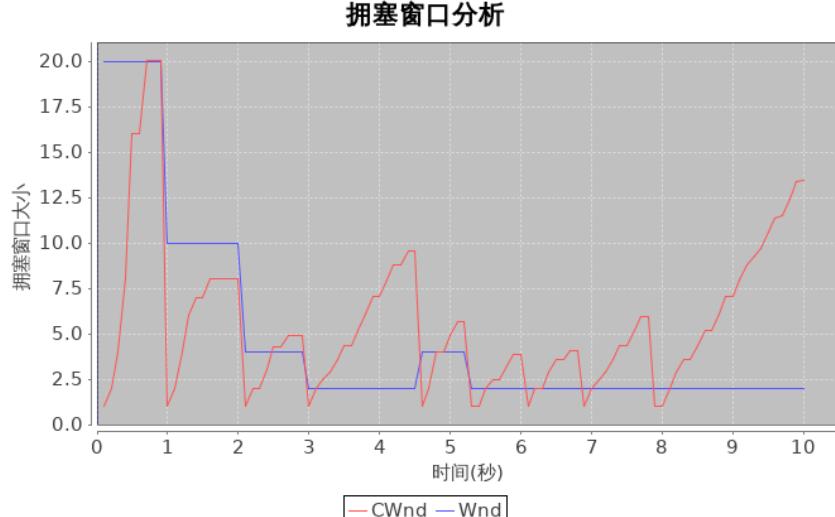
窗口变化图像

拥塞窗口分析

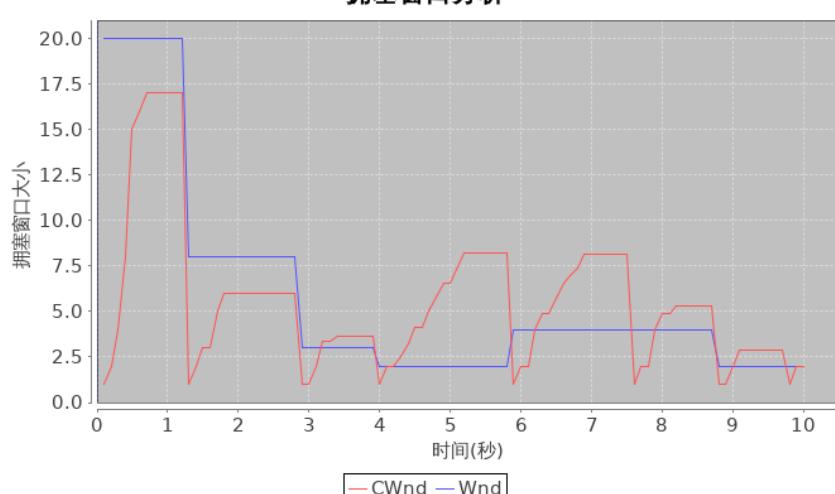
tcp3
Vegas



tcp4
Sack



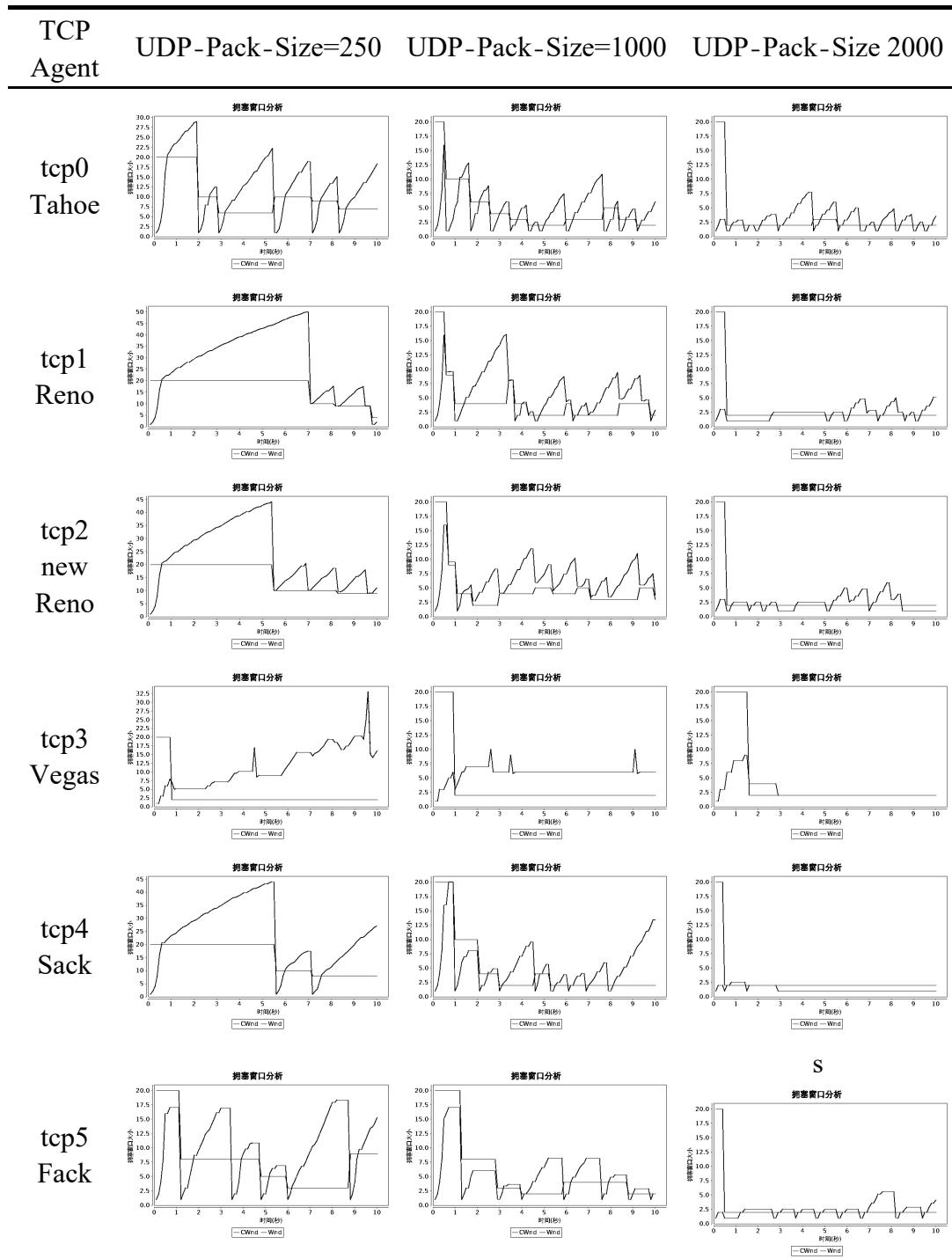
tcp5
Fack



为了更好的对比不同的拥塞控制协议的拥塞控制能力，在不同的网络压力下对这些协议进行模拟，所得到的拥塞控制窗口如表 2-11 所示。其中从左到

右三列 UDP 包大小分别设置为 250、1000 和 2000，分别代表了轻度、中度和重度的网络压力。可以看出在轻度和重度的网络压力下、各个协议都能够很好的工作，除了 Vegas 之外，其余协议窗口变化的震荡幅度均较大，但在重度网络压力下，只有 Tahoe、Reno 和 newReno 勉强能够进行窗口长度的增减，其余协议

表 2-11 不同网络压力下各 Tcp 代理窗口变化曲线



2.3.3 第三项实验的步骤及结果分析

对于第三项实验，在第二项实验的基础上进行修改，其网络结构图与第二项实验相同（见图 2.3）。但将中心链路（n26→n27）上的排队算法分别改为 DropTail、DRR、RED、FQ、SFQ 进行测试，并测试 n26 上的节点收发比特数。模拟结果如表 2-12 所示。

表 2-12 不同网络压力下不同排队协议模拟结果

UDP 包大小 /Bytes	排队协议	总发送字节	总接收字节	带宽利用率
250	DropTail	21272160	21389820	42.66%
	DRR	20024240	20111900	40.14%
	RED	18313300	18422120	36.74%
	FQ	22266460	23008560	45.28%
	SFQ	21018420	21328220	42.35%
1000	DropTail	15261220	15856340	31.12%
	DRR	15588720	16512600	32.10%
	RED	14548140	15153160	29.70%
	FQ	22515180	30473780	52.98%
	SFQ	17076080	19510380	36.58%
2000	DropTail	12984460	21128200	34.11%
	DRR	15518840	26256460	41.78%
	RED	12574620	20261440	32.83%
	FQ	22549720	40455320	63.01%
	SFQ	16381980	27969380	44.35%

从表中可以看处，在不同的网络压力环境下均为公平排队算法 FQ 的带宽利用率最高，而随机早期检测算法 RED 的带宽利用率最低。图 2.4 更为直观的表现它们对于网络的利用状况。

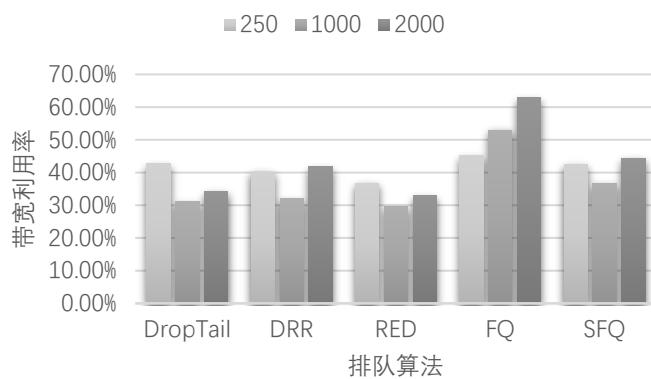
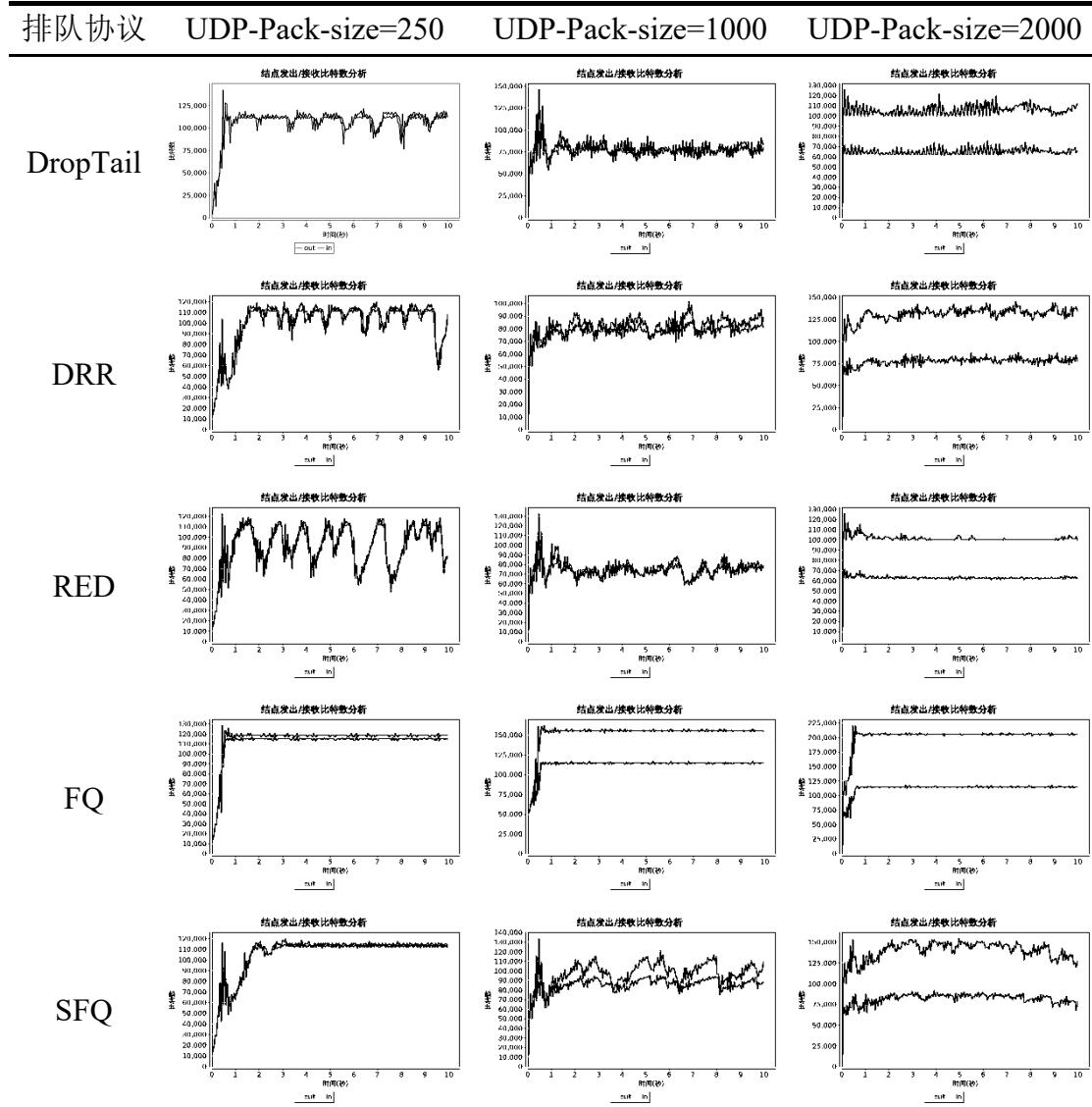


图 2.4 不同排队算法在不同网络压力下的带宽利用率

表 2-13 展示了各个排队协议在不同网络压力下的收发比特数，从图中可以看出，在较低或中等的网络负载情况下，DropTail、DRR、RED 的速率振幅比较大，而在高负载情况下这些协议下的链路速率区域稳定，而 FQ 的网络速率则一致保持着稳定状态，SFQ 则恰恰相反，在较高的压力下速率波动反而较大。DropTail 丢弃尾部的数据包导致发送方重新发送部分数据包，导致了一些速率的波动；在压力较低时，DRR 和 RED 为了实现网络公平而需要牺牲一些带宽利用率，在负载较高时带宽趋满负荷因此稳定下来；公平队列 FQ 的实现则较为简单，在保证基本公平但不是完全公平的情况下保证了最大吞吐，而没有为了网络公平性丢弃一些数据包，因此其速率一致较为稳定；随机公平队列 SFQ 则达到了几乎完全公平，但是没有其它的排队算法效率那么高。

表 2-13 各排队协议在不同网络压力下的收发比特数



2.4 其它需要说明的问题

1. 实验得到的所有曲线图均由 xgraph 生成。
2. 由于部分参数设置可能有些不合理，因此对于实验结果的分析可能有些不准确。

实验三 基于 CPT 的组网实验

3.1 环境

处理器: Intel® Core™ i7-6700HQ CPU @ 2.60GHz 2.59GHz
内存大小: 8.00 GB (7.89GB 可用)
操作系统: Archlinux 64 位, 已于实验前对系统进行滚动更新
第三方软件: Packet Tracer 7.1-1

3.2 实验要求

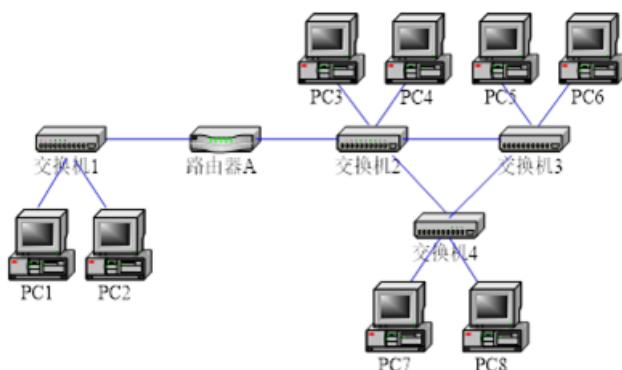


图 3.1 网络拓扑 1

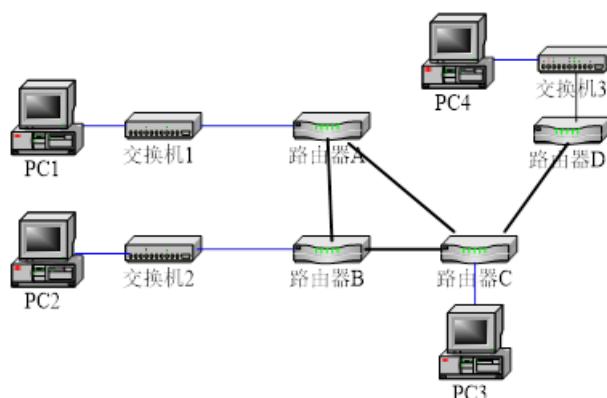


图 3.2 网络拓扑 2

第一项实验——组网实验:

- ✧ 使用仿真软件描述图 3.1。
- ✧ 按照如下要求进行 IP 地址规划:
 - 将 PC1、PC2 设置在同一个网段, 子网地址是: 192.168.0.0,

PC3~PC8 设置在同一个网段，子网地址是：192.168.1.0 同时为路由器配置端口地址，使得两个子网内部的各 PC 机之间可以自由通信。

- 按照如下要求重新设置各 PC 机 IP 地址：
 - ◆ PC1 与 PC2 在一个网段，子网地址是：192.168.0.0；
 - ◆ PC3,PC5,PC7 在一个网段，子网地址是：192.168.1.0；
 - ◆ PC4,PC6,PC8 在一个网段，子网地址是：192.168.2.0；
 - ◆ 为路由器配置端口地址
- 分析各 PC 机之间的连通性并对分析结果进行测试，同时使用所学理论知识对测试结果进行再分析

第二项实验——路由配置实验

- ✧ 使用仿真软件描述网络拓扑图 3.2
- ✧ 按照如下要求配置 RIP 协议：
 - 设置各 PC 机 IP 地址：
 - ◆ PC1 处于 192.168.1.0 网段；
 - ◆ PC2 处于 192.168.2.0 网段；
 - ◆ PC3 处于 192.168.3.0 网段；
 - ◆ PC4 处于 192.168.4.0 网段
 - 设置路由器端口的 IP 地址
 - 在路由器上配置 RIP 协议，使各 PC 机能互相访问
- ✧ 思考题（进阶）
 - 如果不设置时钟频率，会出现什么现象
 - 在路由器上重新配置 OSPF 协议，使各 PC 机能互相访问

第三项实验——VLAN 划分实验：

- ✧ 在第一项实验的最终配置结果上进行 VLAN 划分
- ✧ 划分 VLAN，并按照如下所述配置各 VLAN 的访问权限：
 - 将交换机 2、交换机 3、交换机 4 组成的部分网络（路由器 A 右部网络）划分成 2 个 VLAN：
 - ◆ PC3、PC5、PC7 处于一个 VLAN；
 - ◆ PC4、PC6、PC8 处于一个 VLAN。
 - 测试上述 PC 机之间的连通性。
- ✧ 思考题（进阶）
 - 对路由器 A 进行配置，使得划分的两个 VLAN 中的所有 PC 可

以互相访问

- 将 PC1 和 PC2 划入 VLAN1

第四项实验——访问控制配置实验：

- ✧ 对路由配置实验结果按如下要求进行访问控制配置（ACL）实验
 - 对路由器 1 进行访问控制配置，使得 PC1 无法访问其它 PC，也不能被其它 PC 机访问。
 - ✧ 思考题（进阶）
 - 进行访问控制配置，使得 PC1 不能访问 PC2，但能访问其它 PC 机

综合部分：

某学校申请了一个前缀为 211.69.4/22 的地址块，准备将整个学校连入网络。该学校有 4 个学院，1 个图书馆，3 个学生宿舍。每个学院有 20 台主机，图书馆有 100 台主机，每个学生宿舍拥有 200 台主机。

组网需求：

- ✧ 图书馆能够无线上网
- ✧ 学院之间可以相互访问
- ✧ 学生宿舍之间可以相互访问
- ✧ 学院和学生宿舍之间不能相互访问
- ✧ 学院和学生宿舍皆可访问图书馆。

3.3 基本部分实验步骤说明及结果分析

3.3.1 组网实验的步骤及结果分析

对于组网实验的第一个子实验，首先按照网络拓扑图选择设备并将设备拖拽到主界面中，此处选择的终端设备为台式计算机 PC-PT，路由器型号为 1941，交换机型号为 Switch-PT。本着不同设备之间用直通线，同种设备之间用交叉线的原则，计算机与交换机之间选用直通线，交换机与交换机之间选择交叉线，交换机与路由器之间选用直通线按照拓扑图进行连接。

在连接完成后双击各个计算机并按照要求分别为其分配静态 ip 地址。对于图 3.3 中的设备，PC1 与 PC2 分配的 ip 地址为 192.168.0.101，PC3~PC8 分配的 ip 地址为 192.168.1.100 到 192.168.1.105 的连续地址（PC3 分配 192.168.1.100，

PC4 分配 192.168.1.101, 以此类推)。并将路由器左边的 PC 的 Gateway 配置为 192.168.0.1, 路由器右边 PC 的 Gateway 配置为 192.168.1.1。

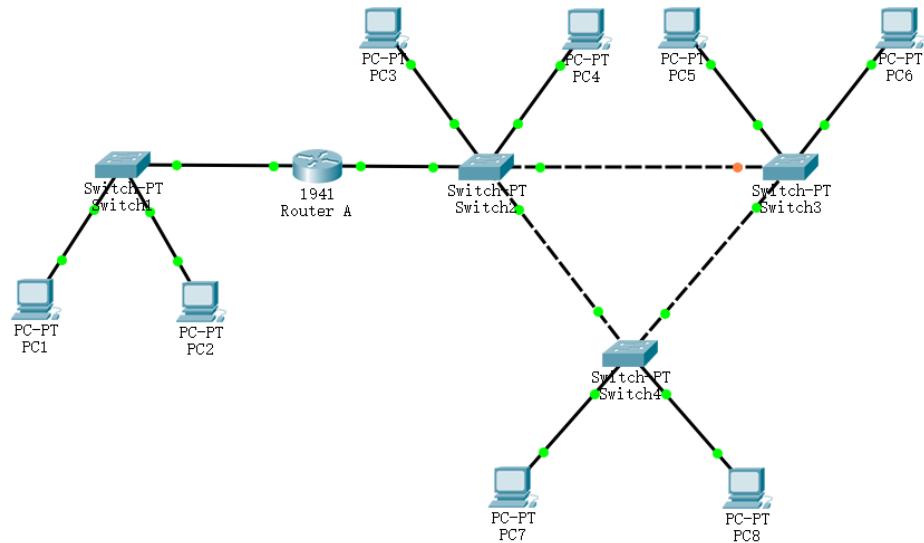


图 3.3 实验 1 连接完成界面

最后, 打开路由器的配置界面, 将路由器左端接口 GigabitEthernet0/0 的 ip 配置为 192.168.0.1, 右端接口 GigabitEthernet0/1 的 ip 配置为 192.168.1.1, 配置完成后如图 3.4 所示。此时, 各个 PC 之间应该可以通信。

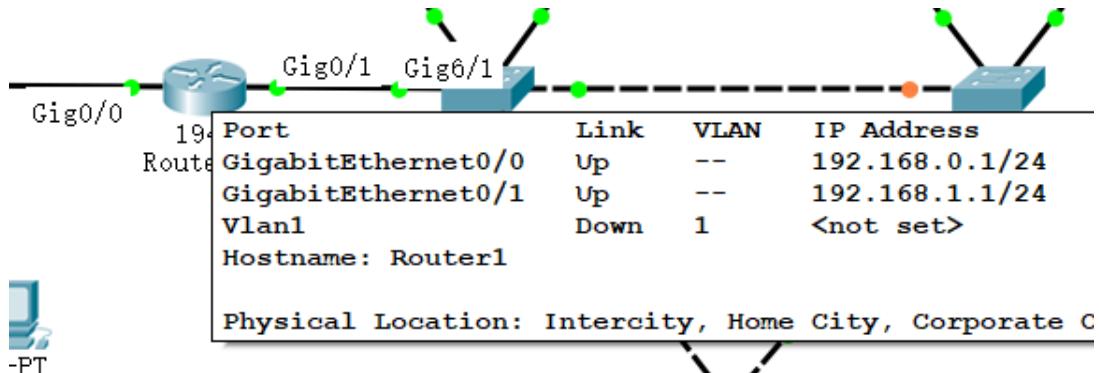


图 3.4 配置完成后的路由器各个接口 ip

回到主界面, 使用 ADD Sample PDU 工具令各个 PC 之间互相通信, 实验结果如图 3.5 所示, 无论是否跨越路由器或交换机, 各个 PC 之间均可以通信。

PDU List Window								
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num
Successful	PC1	PC2	ICMP	blue	0.000	N	0	
Successful	PC1	PC3	ICMP	teal	0.000	N	1	
Successful	PC1	PC5	ICMP	light green	0.000	N	2	
Successful	PC1	PC7	ICMP	medium green	0.000	N	3	
Successful	PC3	PC4	ICMP	magenta	0.000	N	4	
Successful	PC3	PC5	ICMP	light red	0.000	N	5	
Successful	PC3	PC7	ICMP	dark blue	0.000	N	6	
Successful	PC5	PC6	ICMP	magenta	0.000	N	7	
Successful	PC5	PC7	ICMP	red	0.000	N	8	
Successful	PC7	PC8	ICMP	yellow	0.000	N	9	

图 3.5 各 PC 之间互相发送 ICMP 报文结果

对于组网实验的第二个子实验，按照修改各台 PC 的 IP 地址如图 3.7 所示。其中路由器的 ip 地址保持不变。然后使用同样的方法令各台 PC 尝试互相

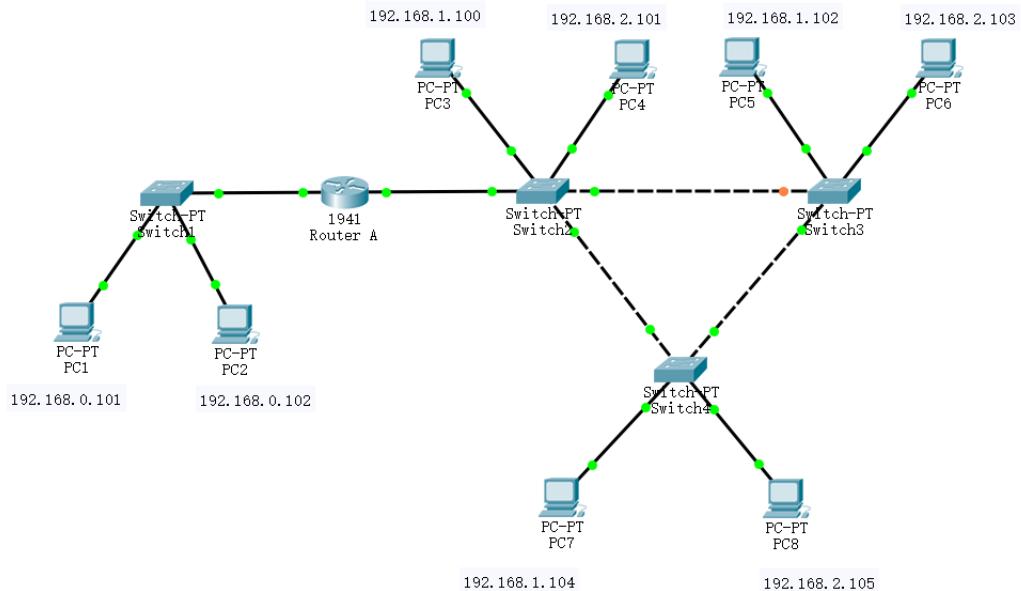


图 3.7 修改后各个 PC 的 ip 地址

通信，通信的结果如图 3.6 所示，其中，PC2，PC4，PC6 在同一个网段中，这一网段中的任一两台机器可以通信，但是 PC2，PC4，PC6 与其他机器的通信均不能成功，因为他们不属于同一个子网且 PC2，PC4，PC4 与路由器右端的接口不处于同一个子网中，因而不能通过 arp 找到对方的 mac 地址，从而不能通信。除了 192.168.2.0 子网中的机器外，其它的机器均可以互相通信成功。

PDU List Window										
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num		
●	Successful	PC1	PC2	ICMP	■	0.000	N	0		
●	Successful	PC1	PC3	ICMP	■	0.000	N	1		
●	Successful	PC1	PC5	ICMP	■	0.000	N	2		
●	Successful	PC1	PC7	ICMP	■	0.000	N	3		
●	Failed	PC1	PC4	ICMP	■	0.000	N	4		
●	Failed	PC1	PC6	ICMP	■	0.000	N	5		
●	Failed	PC1	PC8	ICMP	■	0.000	N	6		
●	Failed	PC3	PC4	ICMP	■	0.000	N	7		
●	Successful	PC4	PC6	ICMP	■	0.000	N	8		
●	Successful	PC6	PC8	ICMP	■	0.000	N	9		

图 3.6 重新划分子网后互相通信结果

3.3.2 路由配置实验的步骤及结果分析

对于路由配置实验，类似于组网实验，首先将各个设备拖到主屏幕上，选取的设备型号与路由配置实验相同。较为特殊的一点是路由器之间的连接使用

的串口 DCE/DTE 连线，并且 PC 与路由器之间的连线为交叉线。连接好后的图如图 3.8 所示。然后配置每个 PC1 的 ip 地址。PC1~PC4 的 IP 地址分别为 192.168.1.100、192.168.2.100、192.168.3.100、192.168.4.100。然后配置路由器的端口地址。

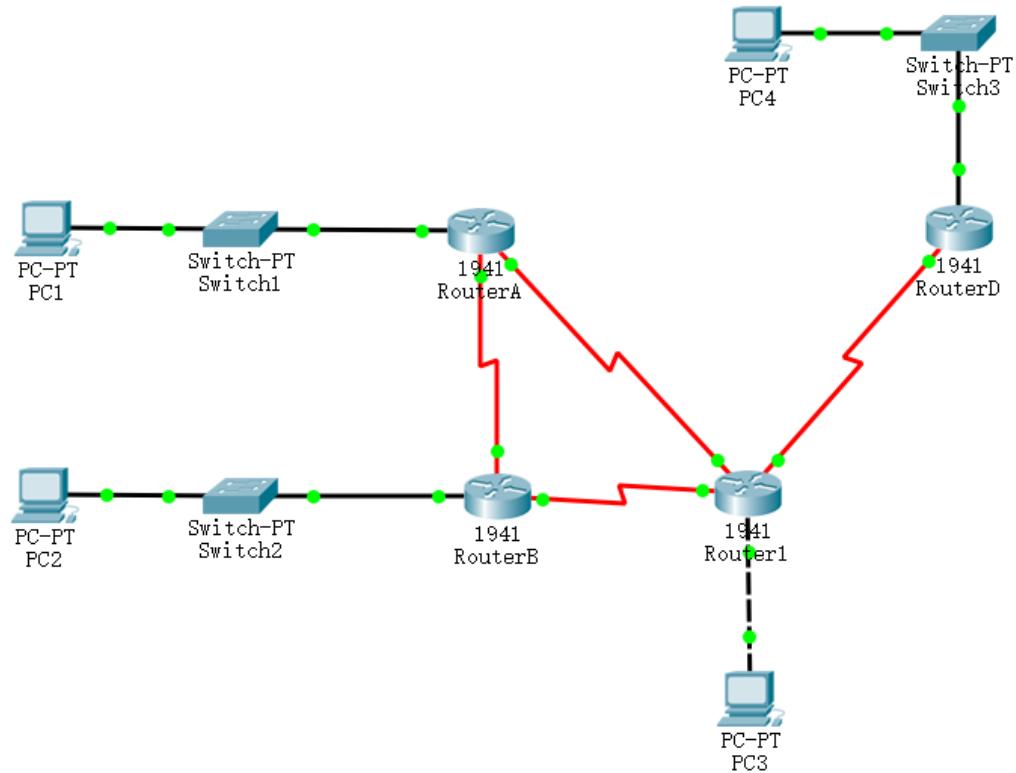


图 3.8 路由配置实验连线

连线完成后，图中 RouterA 与 Switch1 连接的接口 ip 为 192.168.1.1，RouterB 与 Switch2 连接的接口地址为 192.168.2.1，Router1 与 PC3 连接的接口 ip 为 192.168.3.1，RouterD 与 Switch3 连接的接口 ip 为 192.168.4.1。Router1 之所以命名较为特殊，是因为他与其它三个路由器均有连接，且均为串口的 DCE（时钟）端。接下来配置路由器之间的端口。Router1 与 RouterA、RouterB、RouterD 的连接接口地址分别为 192.168.0.1、192.168.0.5、192.168.0.9，对端的接口 ip 地址为 192.168.0.2、192.168.0.6、192.168.0.10。此外。RouterA 与 RouterB 连接的接口地址为 192.168.0.13、对端的接口 ip 地址为 192.168.0.14。

然后在各个路由器上配置始终频率，使各个路由器之间的时钟频率均为 9600。最后配置 RIP 协议，双击路由器后再 RIP 配置界面把各个路由器自己已知的子网地址填入其中即可。最后测试通信状态，四台 PC 之间两两均能通信，如图 3.9 所示。

PDU List Window									
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	
●	Successful	PC4	PC1	ICMP	■	0.000	N	0	
●	Successful	PC4	PC2	ICMP	■	0.000	N	1	
●	Successful	PC4	PC3	ICMP	■	0.000	N	2	
●	Successful	PC1	PC4	ICMP	■	0.000	N	3	
●	Successful	PC1	PC2	ICMP	■	0.000	N	4	
●	Successful	PC1	PC3	ICMP	■	0.000	N	5	
●	Successful	PC2	PC1	ICMP	■	0.000	N	6	
●	Successful	PC2	PC4	ICMP	■	0.000	N	7	
●	Successful	PC2	PC3	ICMP	■	0.000	N	8	
●	Successful	PC3	PC1	ICMP	■	0.000	N	9	
●	Successful	PC3	PC2	ICMP	■	0.000	N	10	
●	Successful	PC3	PC4	ICMP	■	0.000	N	11	

图 3.9 路由配置实验各个 PC 通信结果

对于思考题第一题而言，如果不配置时钟频率，在 Packet Tracer 7.1 中能够正常通信。其实理论上不设置频率时不能正常通信，因为时钟是通信的基础。但是在 Packet Tracer 7.1 版本中即使将时钟频率设置为 not set 程序也会自动将时钟频率设置为 2000000 以保证其正常通信。

对于思考题第二题而言，ospf 协议需要一个主干区域，其它所有区域都必须直接连接到主干区域上。最终区域划分如图 3.10 所示其中 RouterB 与 Router1 连接的区域被定为主干区域，其它区域都与之连接。然后打开路由器的 CLI 界

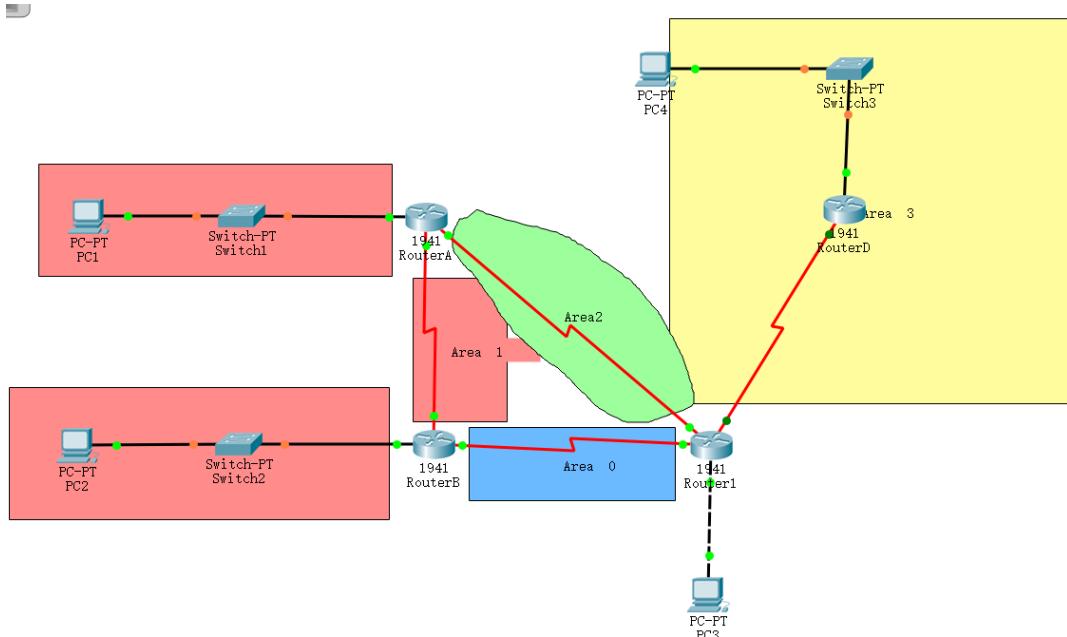


图 3.10 ospf 区域划分

面（ospf 只能在 CLI 界面配置）进行配置。配置完成后结果如图 3.11 所示（展示的为 Router1 的 startup-config。可以看到 ospf 字段对于各个区域的划分。划

```

Router1
Physical Config CLI Attributes
IOS Command Line Interface
clock rate 2000000
!
interface Serial0/1/1
no ip address
clock rate 2000000
shutdown
!
interface Vlan1
no ip address
shutdown
!
router ospf 1
log-adjacency-changes
network 192.168.0.4 0.0.0.3 area 0
network 192.168.3.0 0.0.0.255 area 0
network 192.168.0.0 0.0.0.3 area 2
network 192.168.0.8 0.0.0.3 area 3
!
ip classless
!
ip flow-export version 9
!
!
!--More-- |

```

Ctrl+F6 to exit CLI focus

Top

图 3.11 Router1 的配置

分完成后使用与基础部分相同的方法测试各个 PC 之间是否能够通信。测试结果表明各个 PC 之间两两能够通信，如图 3.12 所示，因而可以认为 ospf 协议配置成功。

PDU List Window									
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	
●	Successful	PC4	PC1	ICMP	■	0.000	N	0	
●	Successful	PC4	PC2	ICMP	■	0.000	N	1	
●	Successful	PC4	PC3	ICMP	■	0.000	N	2	
●	Successful	PC1	PC4	ICMP	■	0.000	N	3	
●	Successful	PC1	PC2	ICMP	■	0.000	N	4	
●	Successful	PC1	PC3	ICMP	■	0.000	N	5	
●	Successful	PC2	PC1	ICMP	■	0.000	N	6	
●	Successful	PC2	PC4	ICMP	■	0.000	N	7	
●	Successful	PC2	PC3	ICMP	■	0.000	N	8	
●	Successful	PC3	PC1	ICMP	■	0.000	N	9	
●	Successful	PC3	PC2	ICMP	■	0.000	N	10	
●	Successful	PC3	PC4	ICMP	■	0.000	N	11	

图 3.12 配置 ospf 后各个 PC 之间的通信关系

3.3.3 VLAN 划分实验的步骤及结果分析

对于实验 1 的最终结果 (ip 地址分配如图 3.7 所示) 进行 vlan 划分, 划分方式为双击路由器 (以 Switch2 为例), 然后在路由器的 Vlan Database 中进行配置, 如图 3.13 所示, 其中 PC3、PC5、PC7 被分配在编号为 2 的 oddPC 虚拟

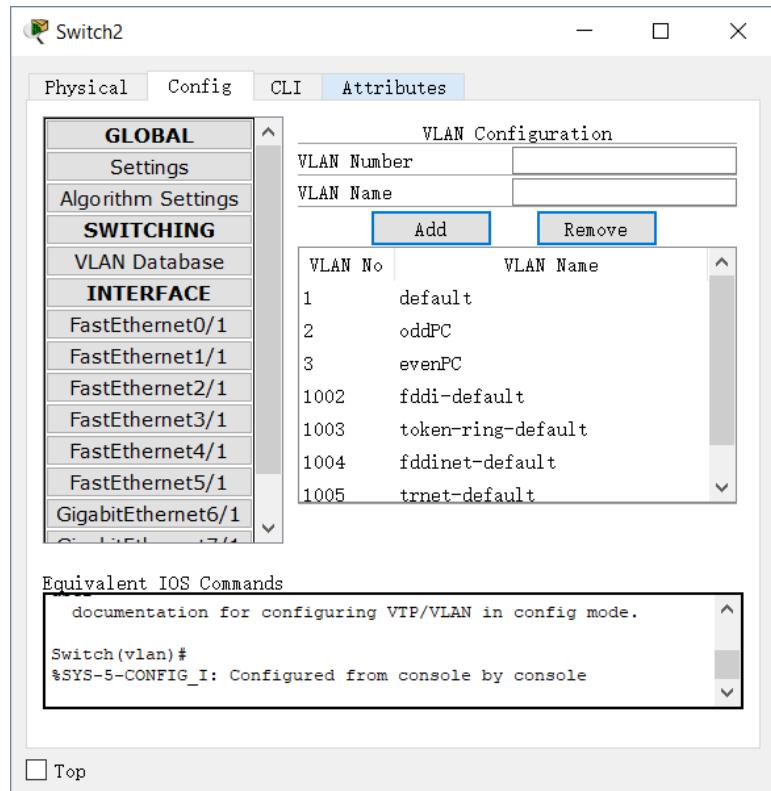


图 3.13 VLAN 配置界面

子网下, 而 PC4、PC6、PC8 被分配在 VLAN 编号为 3 的 evenPC 下, 最后在交换机 interface 配置中选择 access 模式下的 VLAN 编号即可。VLAN 配置成功后网络连通性与实验一重新分配 ip 地址后的网络连通性相同 (见图 3.6), 虽然分配了虚拟局域网, 但是由于 ip 地址的分配没有变, 所以连通性测试结果相同。

对于思考题的第一题而言, 让两个 VLAN 能够互相通信需要一个单臂路由器, 而图中的 RouterA 就承担了这个路由器的角色。对于路由器 A 右端的接口 (GigabitEthernet0/1) 进行配置, 使之有两个子接口 GigabitEthernet0/1.1 和 GigabitEthernet0/1.2, 两个子端口的 ip 分别为 192.168.1.1 和 192.168.2.1, 使他们与两个 VLAN 分别处于两个相同子网中, 并将这一接口的 VLAN 访问模式设置为 Trunk, 配置好后的路由器属性如图 3.14 所示。可以看到 GigabitEhternet0/1 的两个子接口及其地址已经配置。

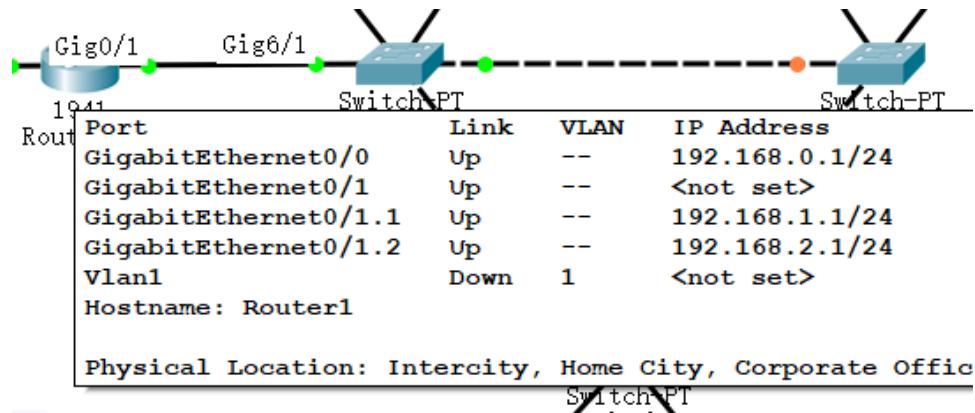


图 3.14 配置好子接口后的路由器端口属性

配置好单臂路由器后，各台 PC 之间两两都能够通信，如图 3.15 所示，尤其值得注意的是，VLAN2 和 VLAN3 两个子网内的各台 PC 可以跨 VLAN 通信。

PDU List Window									
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	
●	Successful	PC1	PC2	ICMP	■	0.000	N	0	
●	Successful	PC1	PC3	ICMP	■	0.000	N	1	
●	Successful	PC1	PC4	ICMP	■	0.000	N	2	
●	Successful	PC1	PC5	ICMP	■	0.000	N	3	
●	Successful	PC1	PC6	ICMP	■	0.000	N	4	
●	Successful	PC1	PC7	ICMP	■	0.000	N	5	
●	Successful	PC1	PC8	ICMP	■	0.000	N	6	
●	Successful	PC3	PC4	ICMP	■	0.000	N	7	
●	Successful	PC3	PC5	ICMP	■	0.000	N	8	
●	Successful	PC6	PC8	ICMP	■	0.000	N	9	

图 3.15 配置好路由器之后的连通性测试

对于思考题的第二题，由于 PC1 与 PC2 默认就在 VLAN1 中，因此不必再特别地进行 VLAN 分配。

3.3.4 访问控制配置实验的步骤及结果分析

对于访问控制实验的第一部分，由于要时 PC1 与外界完全隔离，因此对路

```

interface GigabitEthernet0/0
ip address 192.168.1.1 255.255.255.0
ip access-group 1 in
ip access-group 1 out
duplex auto
speed auto
!
access-list 1 deny 192.168.1.0 0.0.0.255
access-list 1 permit any
!
```

图 3.16 RouterA 的部分配置

由器 A 连接 Switch1 的接口进行配置(见图 3.8),在其上添加一个 access list,其内容为禁止与 PC1 有关的所有访问但允许其它访问,然后在这一接口上应用这一 access list,结果如图 3.16 所示,图中可以看到 access list 的内容以及这一 access list 在 GigabitEthernet0/0 上的应用情况。在应用完成后,对于各个 PC 之间的连通性进行测试,结果如图 3.17 所示。从图中可以看出,所有 PC1 参与的访问均未能通过,而其它机器的访问能够通过,此时已经达到了所需要的效果。

PDU List Window									
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	
●	Failed	PC4	PC1	ICMP	■	0.000	N	0	
●	Successful	PC4	PC2	ICMP	■	0.000	N	1	
●	Successful	PC4	PC3	ICMP	■	0.000	N	2	
●	Failed	PC1	PC4	ICMP	■	0.000	N	3	
●	Failed	PC1	PC2	ICMP	■	0.000	N	4	
●	Failed	PC1	PC3	ICMP	■	0.000	N	5	
●	Failed	PC2	PC1	ICMP	■	0.000	N	6	
●	Successful	PC2	PC4	ICMP	■	0.000	N	7	
●	Successful	PC2	PC3	ICMP	■	0.000	N	8	
●	Failed	PC3	PC1	ICMP	■	0.000	N	9	
●	Successful	PC3	PC2	ICMP	■	0.000	N	10	
●	Successful	PC3	PC4	ICMP	■	0.000	N	11	

图 3.17 访问控制配置后的连通性测试结果

对于思考题而言,由于只需要 PC1 不能访问 PC2 这一条规则,因此在 RouterB 连接

Switch2 的端口上进行配置(PC2 所在的子网),配制过程类似于上一题,配置完成后

RouterB 连接 Switch2 的端口配置如

```
interface GigabitEthernet0/0
ip address 192.168.2.1 255.255.255.0
ip access-group 1 in
ip access-group 1 out
duplex auto
speed auto
!
```

图 3.18 配置完成过后的 RouterB 的端口配置

```
interface GigabitEthernet0/0
ip address 192.168.2.1 255.255.255.0
ip access-group 1 in
ip access-group 1 out
duplex auto
speed auto
!
```

图 3.18 所示。其中 access grop 1 的内容与上一题完全一致,即将 PC1 禁掉。配置并应用完成后进行连通性测试,测试的结果如

```

interface GigabitEthernet0/0
  ip address 192.168.2.1 255.255.255.0
  ip access-group 1 in
  ip access-group 1 out
  duplex auto
  speed auto
!

```

图 3.18 所示。可以看出，除了 PC1 与 PC2 之间的互相访问外，其余的访问均可以正常进行，其连通性测试结果如图 3.19 所示。此处为禁止 PC1 的出入流量，实际上，如果只禁止 PC1 的 out 流量时 PC2 的 ICMP 报文是可以发送到 PC1 的，但是由于 PC1 的返回报文由于访问控制不能到达 PC2，因此测试结果与图 3.19 的测试结果是一致的。

PDU List Window									
Fire	Last Status	Source	Destination	Type	Color	Time(sec)	Periodic	Num	
●	Successful	PC4	PC1	ICMP	■	0.000	N	0	
●	Successful	PC4	PC2	ICMP	■	0.000	N	1	
●	Successful	PC4	PC3	ICMP	■	0.000	N	2	
●	Failed	PC1	PC2	ICMP	■	0.000	N	3	
●	Successful	PC1	PC3	ICMP	■	0.000	N	4	
●	Failed	PC2	PC1	ICMP	■	0.000	N	5	
●	Successful	PC2	PC4	ICMP	■	0.000	N	6	
●	Successful	PC2	PC3	ICMP	■	0.000	N	7	
●	Successful	PC3	PC1	ICMP	■	0.000	N	8	
●	Successful	PC3	PC2	ICMP	■	0.000	N	9	
●	Successful	PC3	PC4	ICMP	■	0.000	N	10	
●	Successful	PC1	PC4	ICMP	■	0.000	N	11	

图 3.19 RouterB 访问控制配置后的连通性测试

3.4 综合部分实验设计、实验步骤及结果分析

3.4.1 实验设计

首先考虑 ip 地址的分配和子网的划分。经过一番考虑，最终划分的子网如表 3-1 所示。这是按照各个部门的人数来分配的，给学院和宿舍分别分配 4 个和 3 个子网是为了便于管理，如访问控制、流量控制的分段管理等。整个校园网只使用一个核心路由器。在实际配置过程中可以使用两个或以上的路由器进行冗余备份，以确保在一个路由器故障的情况下另一个路由器还能正常工作，然而 Packet Tracer 并不支持冗余路由协议，因此使用一个路由器进行示意。

表 3-1 ip 以及子网划分

VLAN(名称)	子网	ip 段	可用 ip	分配给
2(Academy1)	211.69.4.0 /27	211.69.4.0 ~ 211.69.4.31	30	学院 1
3(Academy2)	211.69.4.32/27	211.69.4.32 ~ 211.69.4.63	30	学院 2
4(Academy3)	211.69.4.64/27	211.69.4.64 ~ 211.69.4.95	30	学院 3
5(Academy4)	211.69.4.96/27	211.69.4.96 ~ 211.69.4.127	30	学院 4
6(Library)	211.69.4.128/25	211.69.4.128 ~ 211.69.4.255	126	图书馆
7(Dormitory1)	211.69.5.0/24	211.69.5.0 ~ 211.69.5.255	254	宿舍 1
8(Dormitory2)	211.69.6.0/24	211.69.6.0 ~ 211.69.6.255	254	宿舍 2
9(Dormitory3)	211.69.7.0/24	211.69.7.0 ~ 211.69.7.255	254	宿舍 3

对于网络拓扑的分配，考虑到一个交换机大约只能接 20~30 个设备，于是

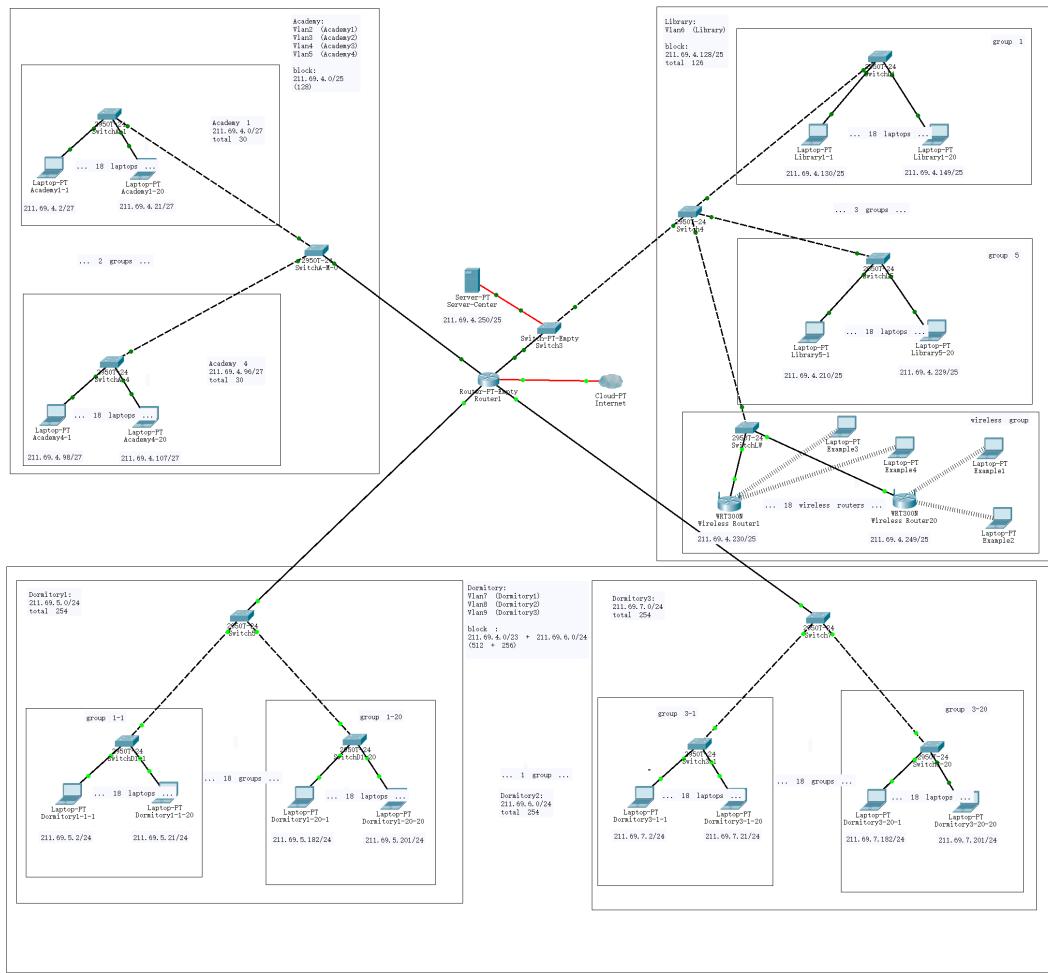


图 3.20 总网络架构

将各个部门的设备以 20 个为一组进行分组，每组分配一个交换机，然后各个组之间用交换机进行树状的连接，整体的网络架构如图 3.20 所示。其中左上部分表示学院区域，右上区域表示图书馆，而下方区域表示学生宿舍。

下面对各个部分放大并逐个进行介绍。以学院 1 为例，其网络架构如图 3.21 所示，由于每个学院只有 20 台计算机，刚好每个学院一个分组，使用一

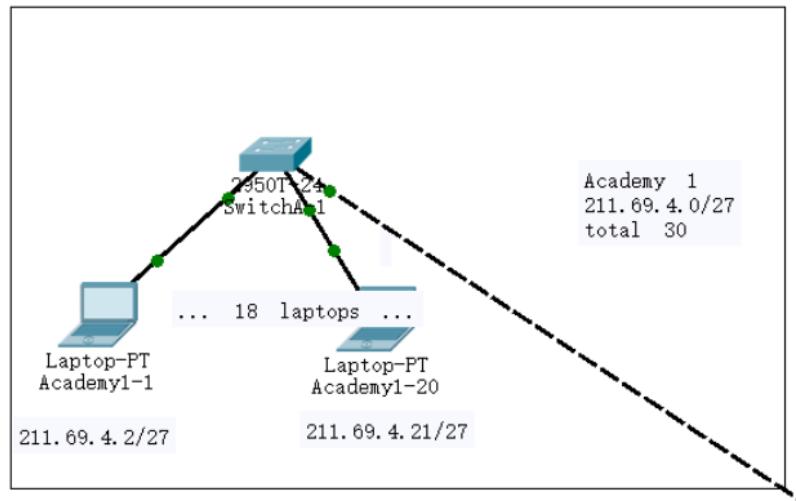


图 3.21 学院 1 的架构

台交换机。图中学院 1 的第一台计算机的 ip 为 211.69.4.2，最后一台计算机的 ip 为 211.69.4.21，共 20 台，由交换机 SwitchA-1 进行连接。四个学院的用于连接各个计算机的交换机分别为 SwitchA-1、SwitchA-2、SwitchA-3、SwitchA-4，这四个交换机由学院的总交换机 SwichA-M-0 进行连接并连接到总路由器上。

对于学生宿舍而言情况有些不同。图 3.22 展示了学生宿舍 1 的网络架构。由于一个学生宿舍有 200 台主机，不能连接在同一个交换机上，因此要进行进

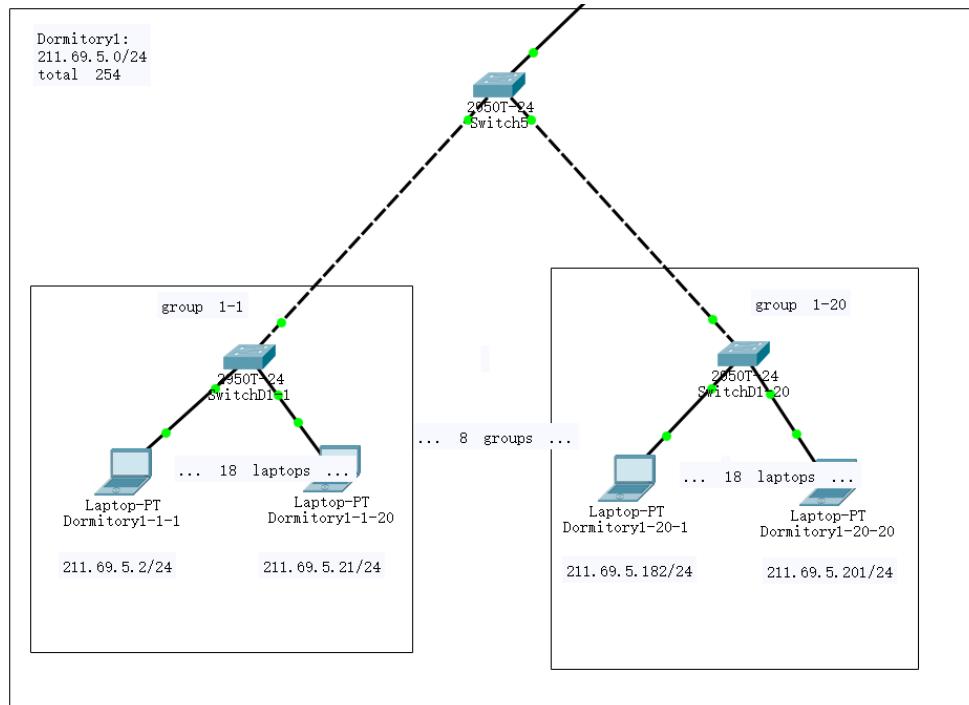


图 3.22 宿舍 1 网络架构

一步的分层：每 20 台机器为一组，连接一个交换机，一个学生宿舍的 10 组的

交换机连接在宿舍的交换机上，三个宿舍的交换机分别连接上学校核心路由器的三个接口。

图书馆的固定台式机的拓扑结构与宿舍类似，每 20 个台式机一组，分配一个交换机，一共 5 组，连在图书馆总交换机上。然而有一点不同的是图书馆总交换机上还连接了另外一个交换机 SwitchLW，这一交换机连接了 20 个无线路由器给图书馆提供无线上网的功能，其网络架构如图 3.23 所示。

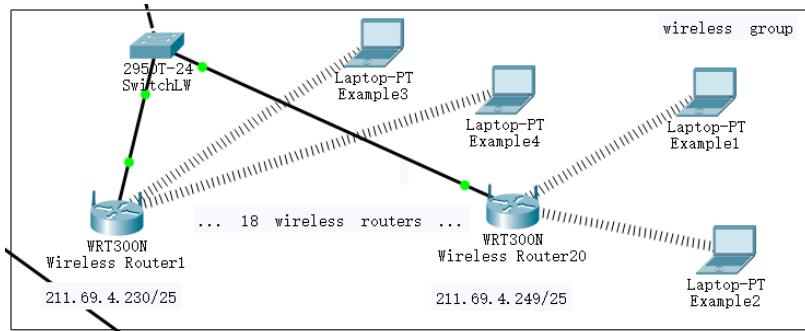


图 3.23 图书馆无线路由器组网络架构

3.4.2 实验步骤

首先按照总的网络拓扑图进行连线，采取的依然是同种设备之间交叉线，异种设备之间直通线的原则。在连线分配完成之后，为各台计算机配置 IP 地址。对于每一组用于表示整个的两个计算机，对其分配这一组的第一的 ip 地址和最后一个 ip 地址。

对于学院的总交换机，学生宿舍的三个总交换机以及图书馆的总交换机划分 VLAN，VLAN 划分如表 3-1 所示。VLAN 划分完成后，对于路由器进行配

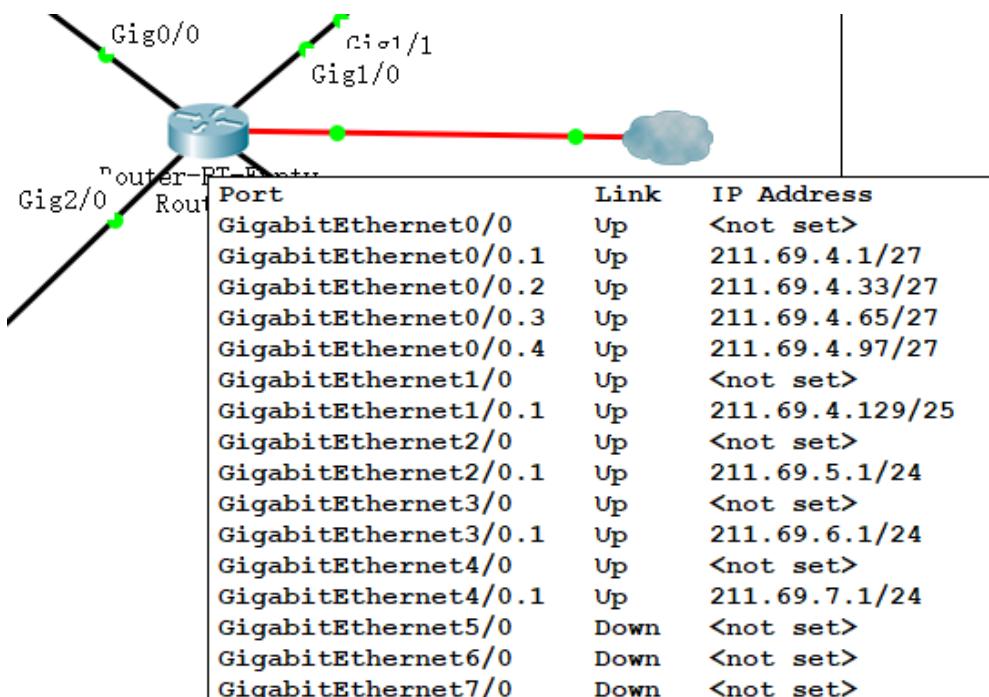


图 3.24 核心路由器配置

置，对其进行子端口的分配、ip 的划分以及 VLAN 的分配。分配方法与 3.3.3 小节相同，配置完成后的结果如图 3.24 所示。其中 GigabitEthernet0/0 连接学院的交换机，GigabitEthernet2/0，GigabitEthernet3/0，GigabitEthernet4/0 分别连接三个宿舍的交换机，GigabitEthernet1/0 连接图书馆的交换机。

最后，在核心路由器上对于各个子网间的访问控制进行配置，配置方法同 3.3.4 小节。最终，路由器的部分配置如下（由路由器上的 running-config 导出并进行截取）。

```
interface GigabitEthernet0/0.1
    encapsulation dot1Q 2
    ip address 211.69.4.1 255.255.255.224
    ip access-group 2 out
!
interface GigabitEthernet0/0.2
    encapsulation dot1Q 3
    ip address 211.69.4.33 255.255.255.224
    ip access-group 2 out
!
interface GigabitEthernet0/0.3
    encapsulation dot1Q 4
    ip address 211.69.4.65 255.255.255.224
    ip access-group 2 out
!
interface GigabitEthernet0/0.4
    encapsulation dot1Q 5
    ip address 211.69.4.97 255.255.255.224
    ip access-group 2 out
!
interface GigabitEthernet1/0.1
    encapsulation dot1Q 6
    ip address 211.69.4.129 255.255.255.128
!
interface GigabitEthernet2/0.1
    encapsulation dot1Q 7
    ip address 211.69.5.1 255.255.255.0
    ip access-group 1 out
```

```
!
interface GigabitEthernet3/0.1
    encapsulation dot1Q 8
    ip address 211.69.6.1 255.255.255.0
    ip access-group 1 out
!
interface GigabitEthernet4/0.1
    encapsulation dot1Q 9
    ip address 211.69.7.1 255.255.255.0
    ip access-group 1 out
!
access-list 1 deny 211.69.4.0 0.0.0.31
access-list 1 deny 211.69.4.32 0.0.0.31
access-list 1 deny 211.69.4.64 0.0.0.31
access-list 1 deny 211.69.4.96 0.0.0.31
access-list 1 permit any
access-list 1 remark 'deny all academy'
access-list 2 deny 211.69.5.0 0.0.0.255
access-list 2 deny 211.69.6.0 0.0.0.255
access-list 2 deny 211.69.7.0 0.0.0.255
access-list 2 permit any
access-list 2 remark 'deny all dormitory'
!
```

最终，对于配置完成的整个网络进行测试。在 PDU list 中创建 7 个 S 从、Scenario，分别用于测试学院内部、宿舍内部、图书馆内部、学院到图书馆、宿舍到图书馆、学院到宿舍、宿舍到学院的访问。各个部分的访问结果如表 3-2 所示。从表中可以看出，除了学院和学生宿舍不能互相访问外，其余部分均可以互相访问，各个部门内部的 PC 也可以互相访问，即达到了题目的要求。

表 3-2 学校各个部分互相访问结果

Scenario	访问方向	访问结果
----------	------	------

Fire	Last Status	Source	Destination	Type	Num	Color
0	Successful	Academy1-1	Academy1-20	ICMP	0	green
1	Successful	Academy1-1	Academy4-20	ICMP	1	light green
2	Successful	Academy1-1	Academy4-1	ICMP	2	dark green
3	Successful	Academy1-20	Academy1-1	ICMP	3	light green
4	Successful	Academy1-20	Academy4-1	ICMP	4	pink
5	Successful	Academy1-20	Academy4-20	ICMP	5	blue
6	Successful	Academy4-1	Academy1-1	ICMP	6	light green
7	Successful	Academy4-1	Academy1-20	ICMP	7	red
8	Successful	Academy4-1	Academy4-20	ICMP	8	teal
9	Successful	Academy4-20	Academy1-1	ICMP	9	dark red
10	Successful	Academy4-20	Academy1-20	ICMP	10	green
11	Successful	Academy4-20	Academy4-1	ICMP	11	dark green

Fire	Last Status	Source	Destination	Type	Num	Color
0	Successful	Dormitory1-1-1	Dormitory1-1-20	ICMP	0	dark purple
1	Successful	Dormitory1-1-1	Dormitory1-20-1	ICMP	1	green
2	Successful	Dormitory1-1-1	Dormitory3-1-1	ICMP	2	dark green
3	Successful	Dormitory1-20-1	Dormitory1-20-20	ICMP	3	pink
4	Successful	Dormitory1-20-1	Dormitory3-1-1	ICMP	4	purple
5	Successful	Dormitory1-20-1	Dormitory3-20-1	ICMP	5	dark purple
6	Successful	Dormitory3-1-1	Dormitory3-1-20	ICMP	6	green
7	Successful	Dormitory3-1-1	Dormitory3-20-1	ICMP	7	blue
8	Successful	Dormitory3-1-1	Dormitory3-20-20	ICMP	8	cyan
9	Successful	Dormitory3-20-1	Dormitory3-20-20	ICMP	9	pink

Fire	Last Status	Source	Destination	Type	Num	Color
0	Successful	Library1-1	Library1-20	ICMP	0	purple
1	Successful	Library1-1	Library5-20	ICMP	1	pink
2	Successful	Library1-1	Library5-1	ICMP	2	yellow
3	Successful	Library5-1	Library5-20	ICMP	3	yellow
4	Successful	Library5-1	Library1-20	ICMP	4	pink
5	Successful	Example3	Library5-1	ICMP	5	red
6	Successful	Example4	Library5-20	ICMP	6	blue
7	Successful	Example1	Library5-20	ICMP	7	yellow
8	Successful	Example2	Library5-1	ICMP	8	green

Fire	Last Status	Source	Destination	Type	Num	Color
0	Successful	Academy1-20	Library1-1	ICMP	0	green
1	Successful	Academy1-1	Library1-20	ICMP	1	green
2	Successful	Academy4-20	Library5-1	ICMP	2	purple
3	Successful	Academy4-1	Library5-20	ICMP	3	pink
4	Successful	Academy4-1	Library5-1	ICMP	4	blue
5	Successful	Academy4-20	Server-Center	ICMP	5	pink

Fire	Last Status	Source	Destination	Type	Num	Color
0	Successful	Dormitory1-1-1	Library5-1	ICMP	0	green
1	Successful	Dormitory1-20-1	Library5-20	ICMP	1	green
2	Successful	Dormitory3-1-20	Library5-1	ICMP	2	blue
3	Successful	Dormitory3-1-1	Library5-20	ICMP	3	cyan
4	Successful	Dormitory3-20-1	Library1-1	ICMP	4	green
5	Successful	Dormitory3-20-20	Library1-20	ICMP	5	blue
6	Successful	Dormitory3-1-1	Library1-1	ICMP	6	dark red
7	Successful	Dormitory3-20-20	Library5-1	ICMP	7	orange

学院 ↓ 宿舍

Fire	Last Status	Source	Destination	Type	Num	Color
●	Failed	Academy1-1	Dormitory1-1-1	ICMP	0	■
●	Failed	Academy1-20	Dormitory1-1-20	ICMP	1	■
●	Failed	Academy4-1	Dormitory3-1-1	ICMP	2	■
●	Failed	Academy4-20	Dormitory3-1-20	ICMP	3	■
●	Failed	Academy1-1	Dormitory3-20-1	ICMP	4	■
●	Failed	Academy1-20	Dormitory3-20-20	ICMP	5	■

宿舍 ↓ 学院

Fire	Last Status	Source	Destination	Type	Num	Color
●	Failed	Dormitory1-1-1	Academy4-1	ICMP	0	■
●	Failed	Dormitory1-1-20	Academy4-20	ICMP	1	■
●	Failed	Dormitory1-20-1	Academy1-1	ICMP	2	■
●	Failed	Dormitory1-20-20	Academy1-20	ICMP	3	■
●	Failed	Dormitory3-1-1	Academy1-20	ICMP	4	■
●	Failed	Dormitory3-1-20	Academy1-1	ICMP	5	■
●	Failed	Dormitory3-20-1	Academy4-20	ICMP	6	■
●	Failed	Dormitory3-20-20	Academy4-1	ICMP	7	■

3.4.3 结果分析

实验的结果是意料之中的，基本部分的四个实验为综合部分的实验打下了坚实的基础，其各个部分的配置方法与基本实验几乎相同。在子网控制中，对于宿舍与学院分别建立了一个 access list，并将这两个 access list 交叉应用到连接对方的接口上，因此学院访问宿舍与宿舍访问学院的流量在路由器处会被截断，但其它部分的流量不受影响，从而能够达到题目的要求。

3.5 其它需要说明的问题

1. 在图书馆进行无线上网的 PC 可以访问到全校其它所有的计算机，而其它计算机不能访问在图书馆无线上网的计算机，是因为无线路由器上的 NAT 隔离。
2. 整体架构图中核心路由上方的机器为学校服务器，在其上配置了 DNS、HTTP、SMTP、FTP 等服务，并可以正常运作，但其不是本实验的重点，因此未对其进行专门的描述。

心得体会与建议

4.1 心得体会

通过这次实验，我对计算机网络从底层到顶层有了更深入的理解，不仅巩固了课堂上学到的知识，还掌握了一些课外的知识。Socket 编程实验让我对于网络底层的通信，协议的设计有了深刻的认识，并能够通过自己的努力设计出一套自己的协议，通过图形界面的方式展示，并通过一些分析工具提升其性能，这些都是是我收益匪浅的；NS2 协议分析实验带领我认识了各个协议的内容、优势以及劣势，更让我为今后的学习打下了基础；而 CPT 组网实验更是是我进一步加深了对于大型网络的构建与配置的理解，从更高的层面对于计算机网络有了新的认识。

4.2 建议

1. 个人认为实验一（Socket 编程实验）规模过大，涉及的内容过多，不宜作为实验。应该减小其规模，或直接作为课设处理。
2. 实验二（基于 NS2 的协议分析实验）其实可以使用 NS3 或其它软件，一方面，连作者本身也承认这不是一个完善的软件，如引文 4-1 所示，另一方面，NSG 的 bug 也较多，且操作比较非人性化，使用较为不便，而 NS3 作为一个正在被积极开发，并且其使用的 Python 脚本也比 NS2 使用的 OTcl 更易于被学生所接受，综上，我认为使用 NS3 更好。

While we have considerable confidence in ns, ns is not a polished and finished product, but the result of an on-going effort of research and development. In particular, bugs in the software are still being discovered and corrected. Users of ns are responsible for verifying for themselves that their simulations are not invalidated by bugs. We are working to help the user with this by significantly expanding and automating the validation tests and demos.

引文 4-1 <https://www.isi.edu/nsnam/ns/> 的部分引用