**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 组号： | 2-1 |  |  |  |  |
| 姓名： | 李程浩 | 学号： | 2186114147 | 班级： | 计算机86 |
| 姓名： | 毛弘卿 | 学号： | 2183310802 | 班级： | 计算机86 |
|  |  |  |  |  |  |

1. 实验名称

实现一个简单的聊天程序

1. 实验原理

TCP：一个可靠的、有序的、字节流式的传输层端到端协议；

UDP：一个不可靠的、无序的、以报文为单位的传输层协议；

HTTP：基于TCP的、请求-应答式的应用层协议；

WebSocket：基于TCP和HTTP的、全双工的应用层协议。相比HTTP，它弥补了“服务器不能推消息”的缺陷；而相比TCP，它做了封包和简单的消息类型等功能，并且便于web技术使用。

WebRTC：名称源自网页即时通信的缩写，是一个支持网页浏览器进行实时语音对话或视频对话的协议族。

1. 实验目的

1) 掌握 Sockets 的相关基础知识，学习 Sockets 编程的基本函数和模式、框架。

2) 掌握 UDP、TCP 协议及 Client/Server 和 P2P 两种模式的通信原理。

3) 掌握 Socket 编程框架

1. 实验内容
   1. 基本功能

A. 验证用户登录；B. 两个用户的文字聊天；C. 用户之间传输文件，包括二进制的大文件（比如 100MB）

* 1. 高级功能

A. 离线消息；B. 会话——消息的转发域；C. 文件断点续传；D. 基于P2P的视频传输。

1. 实验实现
2. 人员分工

李程浩：服务器端，包括服务器端程序、数据库设计和实现；以及大部分的网络协议设计。

毛弘卿：客户端，UI设计与实现、客户端网络处理等；尤其地，p2p视频功能和文件传输的主要劳动量在客户端方面。

开始时，两人各自测试自己代码；在前后端联调联试阶段，两人共同调试。

实验报告由两人分别撰写各自负责的部分。WS、HTTP协议由李程浩分析；WebRTC协议和p2p原理，由毛弘卿分析。

1. 实验设计
2. 协议

我们的聊天程序：

在**协议**上，综合使用了HTTP、WebSocket、WebRTC协议；

在**网络模式**上，大部分功能使用C/S模式，而视频传输使用p2p模式；

在**内容**上，主要使用json格式，以字符方式组织负载，除了文件传输直接发送二进制位流。

由于我们两人分别负责前后端，而前后端使用网络相互通信。为了更高效地分工协作，我们先行商议、设计了网络协议，并产生了一个网络协议文档（protocol.md）；此后我们两人就可以分别依照此文档，设计实现各自的部分，而彼此的进度互不干扰。具体网络设计已在此文档中展示清楚，此处主要解释我们为何这样选择，以及我们对于我们所使用的协议的理解。

**HTTP协议：**

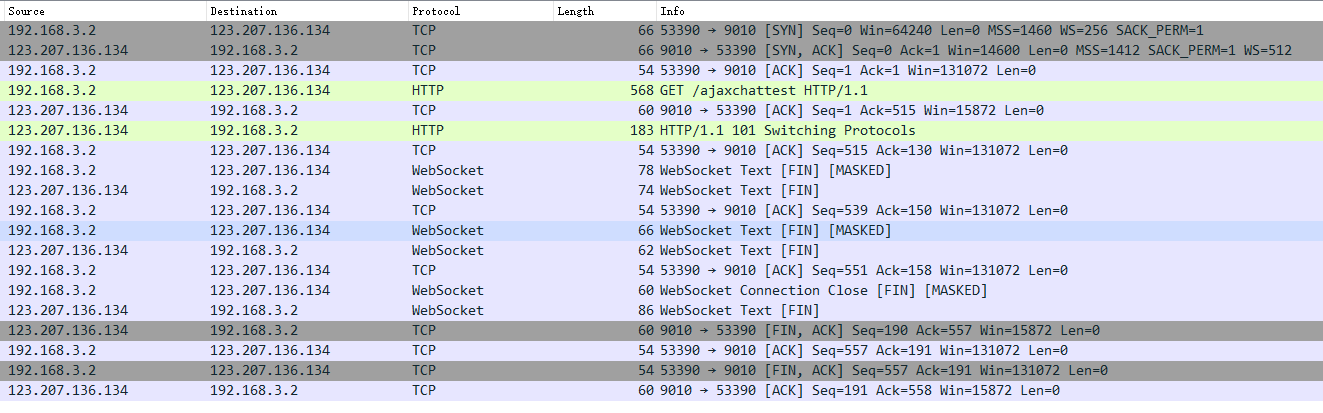
HTTP是著名的应用层协议。我们选择它，一方面是因为它是“**无状态**”的协议，只存留少量状态，可以有效简化我们的程序逻辑；另一方面，HTTP具有“**路由**”的设计，即URL的路径部分，通过向Web框架注册不同的路由，即可有效区分不同的功能，在我们功能数目较多时，有所帮助。

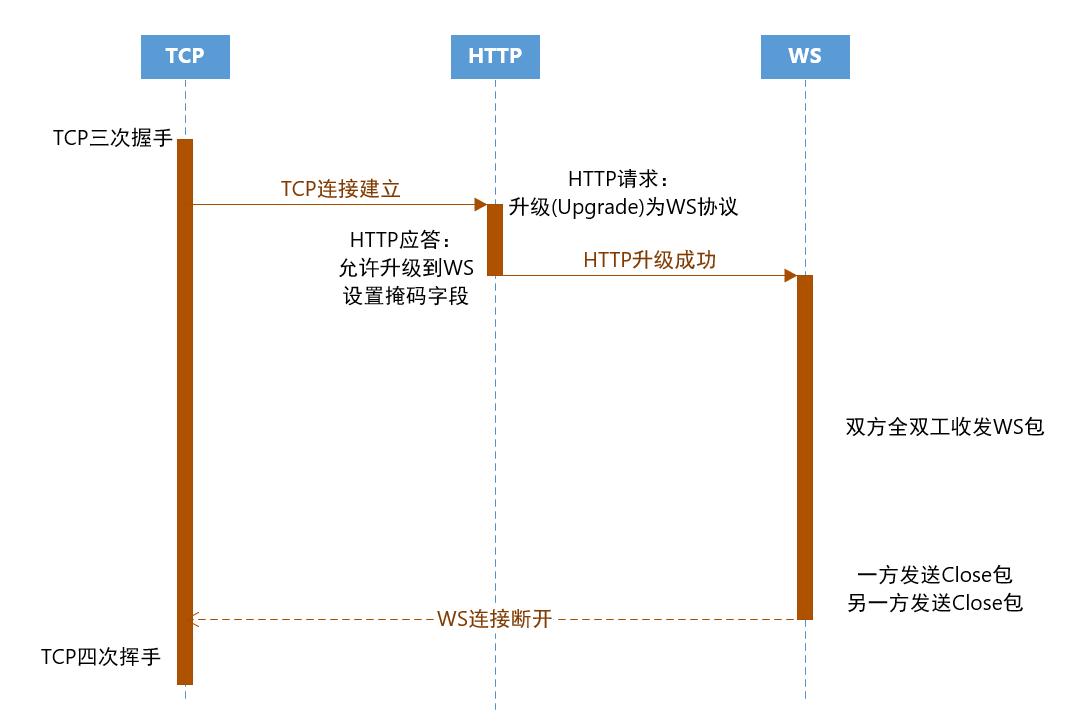
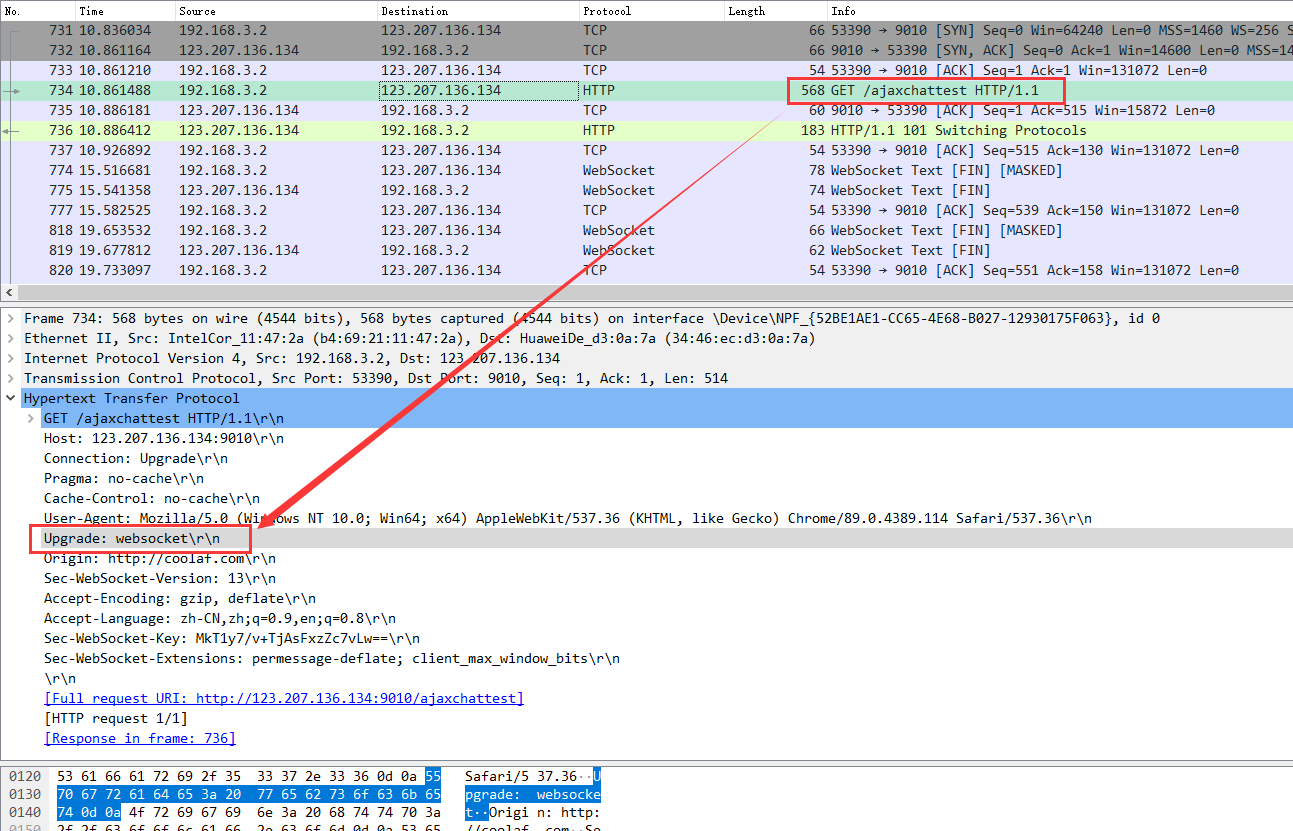
**WebSocket协议：**

HTTP采用C/S间“请求-应答”的方式传输数据，但服务器向客户端“推送”数据的需求仍然存在。在此之前，Web程序员通常采用“轮询”或者“服务器阻塞请求”的方式来实现推送需求，但这样会有性能问题；而WebSocket就是一种全双工的长连接协议，解决了上述问题。

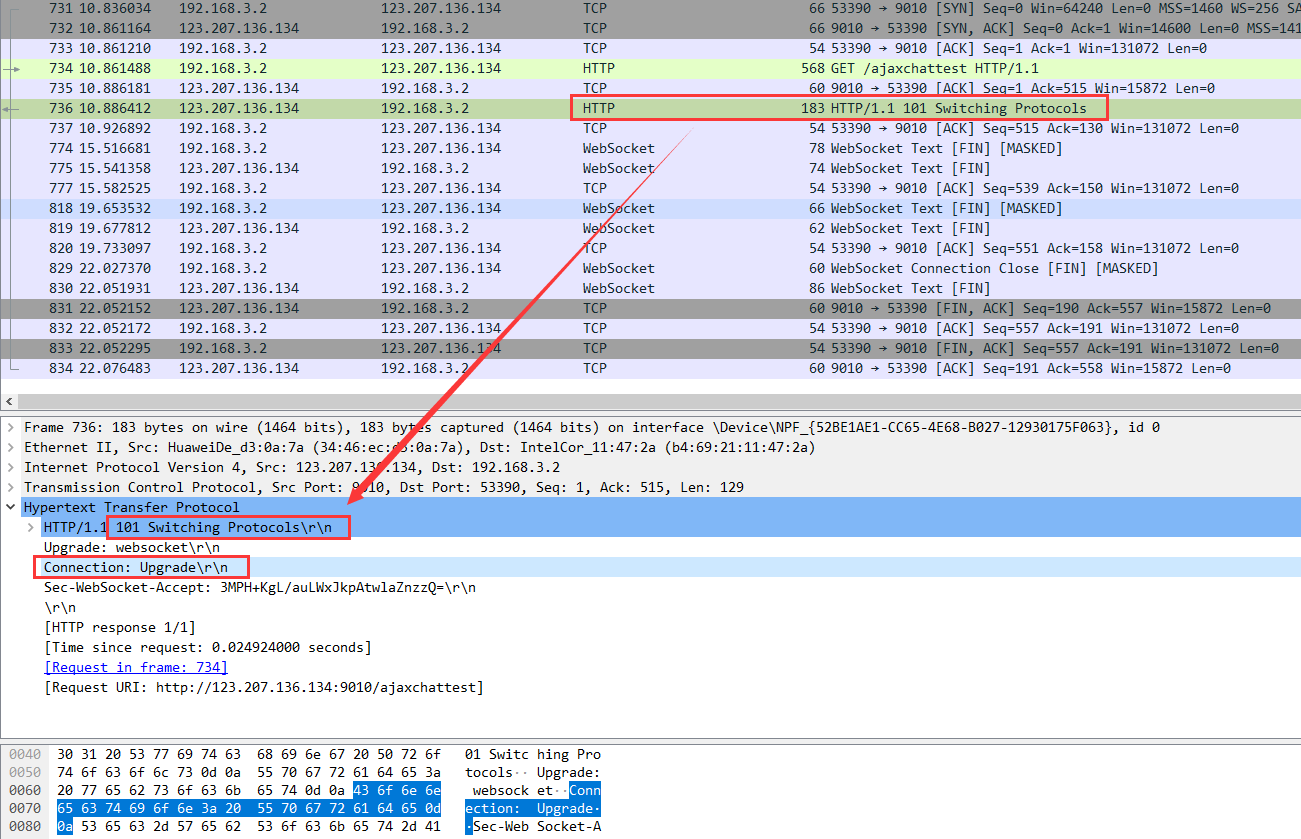
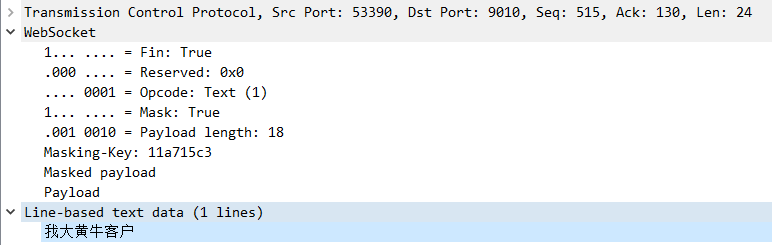
我们的聊天程序有大量“推送”的需求，例如用户A加B为好友，则服务器要“推送”此好友请求到B；消息的转发则更为普遍：当A给会话1中发消息，ta实际上把消息发送给了服务器，而服务器则要把这条消息“推送”给会话1中的所有其它在线成员。并且，我们前端使用Node.js技术，其收发WS异常方便。因此，我们选择使用WebSocket。

而相比于基于字节流的TCP协议，WebSocket协议已经做好了数据的分包、消息的基本类型（字符？二进制？）功能，这对于工期只有2人×月的我们来说，很有帮助。但我们并非逃避实验要求，而是在充分理解、测试了ws协议之后才选择使用它。接下来我就将解析WebSocket协议的字段以及行为：

**行为：**

 以上我所绘制的图就是WS协议的行为，下面是**字段**：

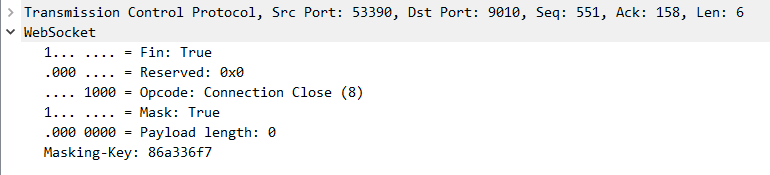
在TCP三次握手成功后，客户端向服务器发送一个HTTP请求，此请求最重要的就是Upgrade: websocket字段，它向服务器请求，把此HTTP所使用的TCP连接，升级为WS连接。

 服务器返回HTTP应答，表示升级协议成功/失败，上图为成功。

上图为一个WS包的字段。可以看到，WS基于TCP，就继承了TCP的所有基本功能，如可靠性、顺序性。

Fin字段类似于IP协议的MF（More Fragments）字段，但意义相反：若IP中的MF段为1，则说明此IP报文经历了拆分，此报文后还有报文，接收方应将其合并；而若WS中的FIN字段为1，则代表此WS报文是最后一个报文。这两个字段都是为了能够突破底层协议对字段长度的限制：若要发送的报文超过了底层协议的限制（如TCP的65535），那就分成多个底层包发送，然后由接收方根据Fin值进行合并。（相比IP，WS不需要段偏移字段，因为底层的TCP协议能保证WS包有序无错传递）

Opcode，4位，由发送者设置，表示WS负载的类型。目前有1：文本、2：二进制、8：WS的Close包、9：心跳包发送包、10：心跳包应答包。

 Payload length字段，有可能是7位、7+16位、7+64位。其变长设计，可以保证在WS包较短时它占用的位数较少，而当WS包很长时也不会超出它的表示范围。

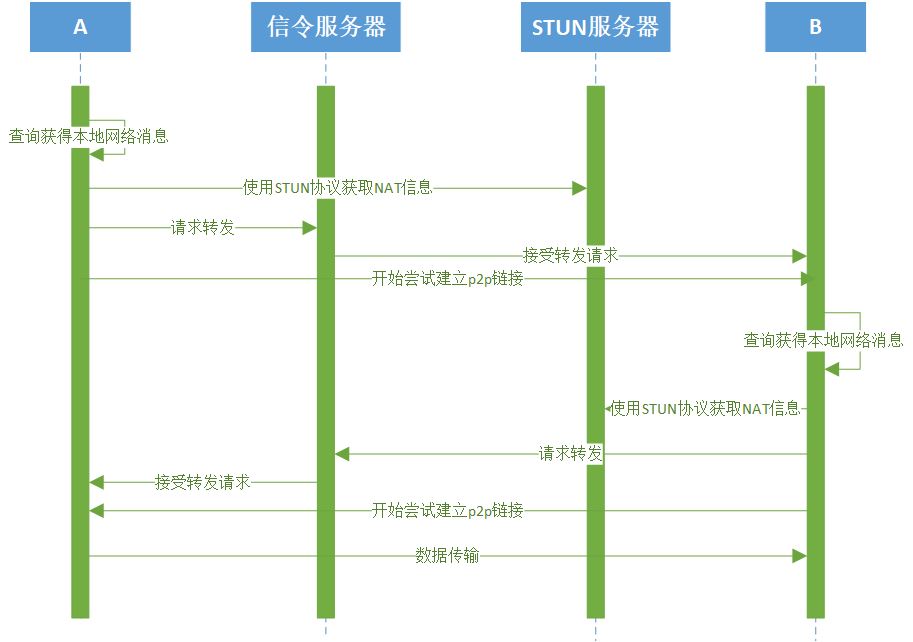
上图即为一个Close包。若一方发送了Close包，则它已经准备关闭此WS连接了，等待对方也发送Close包或超时，他就会释放本地WS连接的缓冲区等，并开始执行TCP的四次挥手。

从上面的分析看，WS相较于TCP，主要实现了分包功能。但其余功能仍然需要我们自己实现。

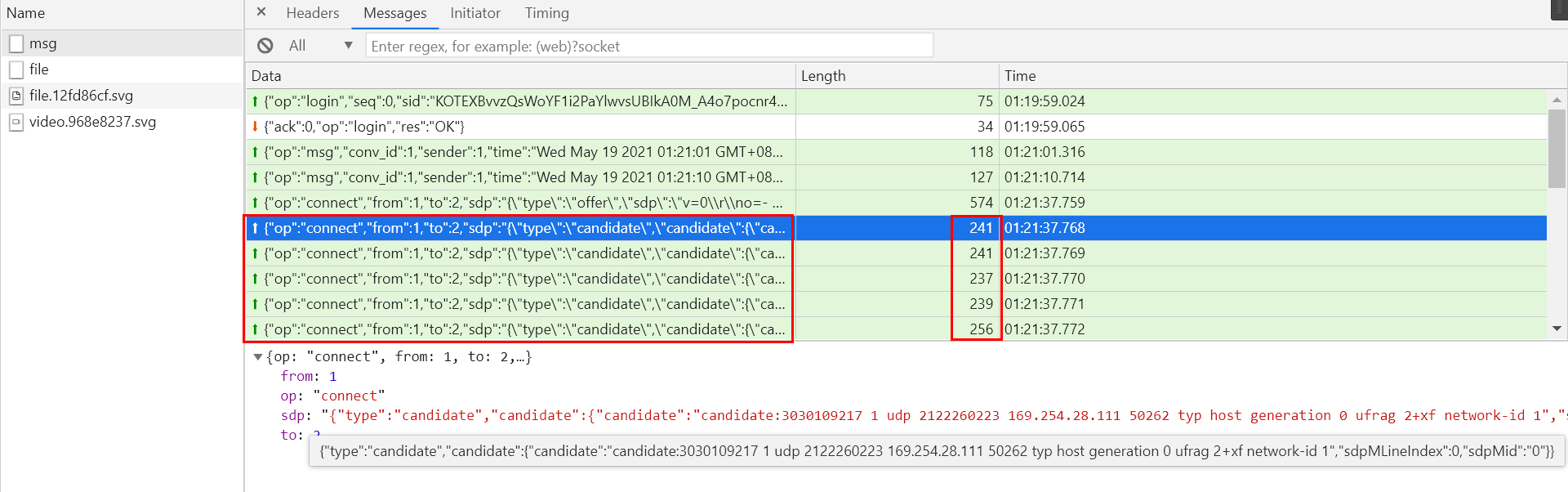
**WebRTC：**

通过stun服务器查询得到本地外网ip和端口，通过本地网络获取本地拘于ip和端口，将信息打包发送给中转服务器（信令服务器）；由中转服务器经过处理（判断目标用户是否在线等）发送给目标用户。实际上，信令服务器不仅需要辅助交换网络信息，还需要传递**业务信息，媒体信息**！

在传输层协议选择上，由于视音频不需要较高的服务质量，但是对服务的延时较为敏感，故选择使用UDP作为底层协议

在实际上，为了节省服务器的带宽资源，建立p2p时会先从局域网开始尝试，如果失败则逐级向上层继续尝试。在最糟糕的情况下，NAT不允许穿透，则使用TURN转发服务器直接转发流量，这种情况下，对转发服务器会产生极大的负载压力（双倍上下行流量），所以在生产环境中可以使用集群+负载均衡的策略通过横向拓展降低对单个服务器的压力，提高服务质量。

在完成p2p连接后，由于建立的是udp通道，如果在NAT传统的情况下，与传统UDP打洞一致需要使用心跳包来保证p2p通道不会被NAT回收，导致连接断开，这一点将由WebRTC协议维护。

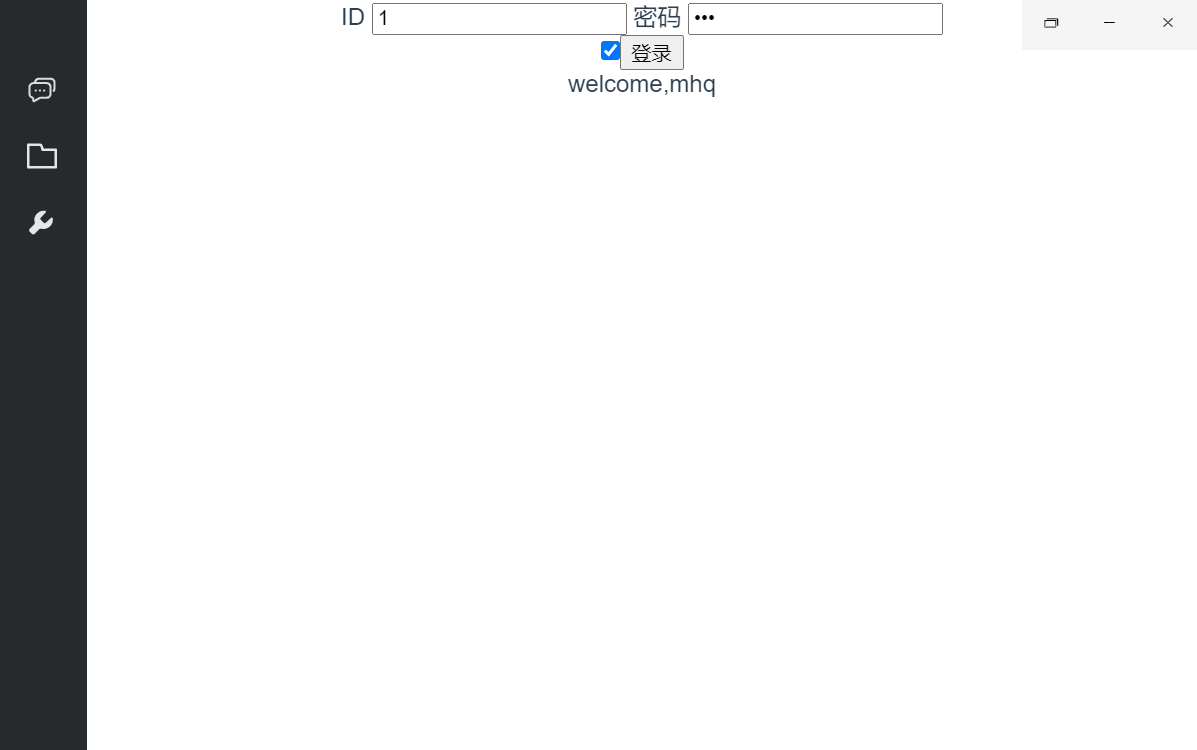
在debug的抓包中可以看到，sdp中包含了协议端口ip等信息，通过**多次发送不同SDP**，将网络环境完整传送给对方

1. UI设计

UI设计整体上以桌面版微信为蓝本，分为登录、聊天和文件传输两个子界面。其中，聊天和文件传输界面中包含了聊天和文件传输的功能。

聊天和文件传输界面分为好友列表和聊天框两个主体，十分清晰地展现了作为一个聊天软件应有的功能，大大降低了用户的上手难度。

好友列表中可以选择聊天的对象。点击聊天和文件可以在聊天模式和文件传输模式之间切换。

聊天模式中可以输入文本（支持UNICODE和表情包）并发送，也可以选择开启视频通话，与对方视频通话；文件传输模式中可以选择文件并发送，界面中会显示文件大小和文件传输进度（百分比），也会在此界面中显示接受的文件和接受进度。



1. 框架结构

**服务器端：**

**Go简介：**

服务器程序使用Golang编写。Go特别适合于并发和网络编程：

1. 使用goroutine替代线程、进程。新建一个goroutine只需要写

*go 函数名（函数参数）* （go是语言的关键字）

goroutine是由用户态调度的，并且与内核线程是m:n的关系；

Goroutine具有动态增长的栈，初始时仅8k，这使得创建百万个goroutine成为可能，而线程则不能创建太多，需要以线程池方式使用；

2. 网络库是同步形式的，并且允许并发调用，则程序员是以多线程+同步形式，进行网络编程，心智负担较小；但编译器会将其代码编译为合适的网络调用方式，比如可能是多线程+每个线程内的IO多路复用，取决于编译器对程序网络调用的分析，以及所处的操作系统。

3. 提供了完善的并发控制工具，如互斥锁sync.Mutex，多读单写锁sync.RWMutex，以及语言语法所支持的特性“通道”（channel）：

*ch <- v // 将 v 发送至信道 ch。*

*v := <-ch // 从 ch 接收值并赋予 v。*

Channel类似于类Unix系统中的“管道”，分为无缓冲的Channel和带缓冲区的Channel：

当一个goroutine向无缓冲的Channel发送数据时，它会阻塞，直到有其它goroutine从这个Channel读数据；读同理；

当一个goroutine向有缓冲的Channel发送数据时，若缓冲已满，它会被阻塞；当一个goroutine从有缓冲的Channel读数据时，若缓冲为空，它会被阻塞。

Channel既可以用来在goroutine之间**传递数据**，也可以用来做goroutine的**同步**；而互斥锁则经常用于**保护某个数据**不被并发访问，或创建一段**临界区域**使得同一时间最多只有一个goroutine能够进入这段临界区域。

**服务器端架构：**

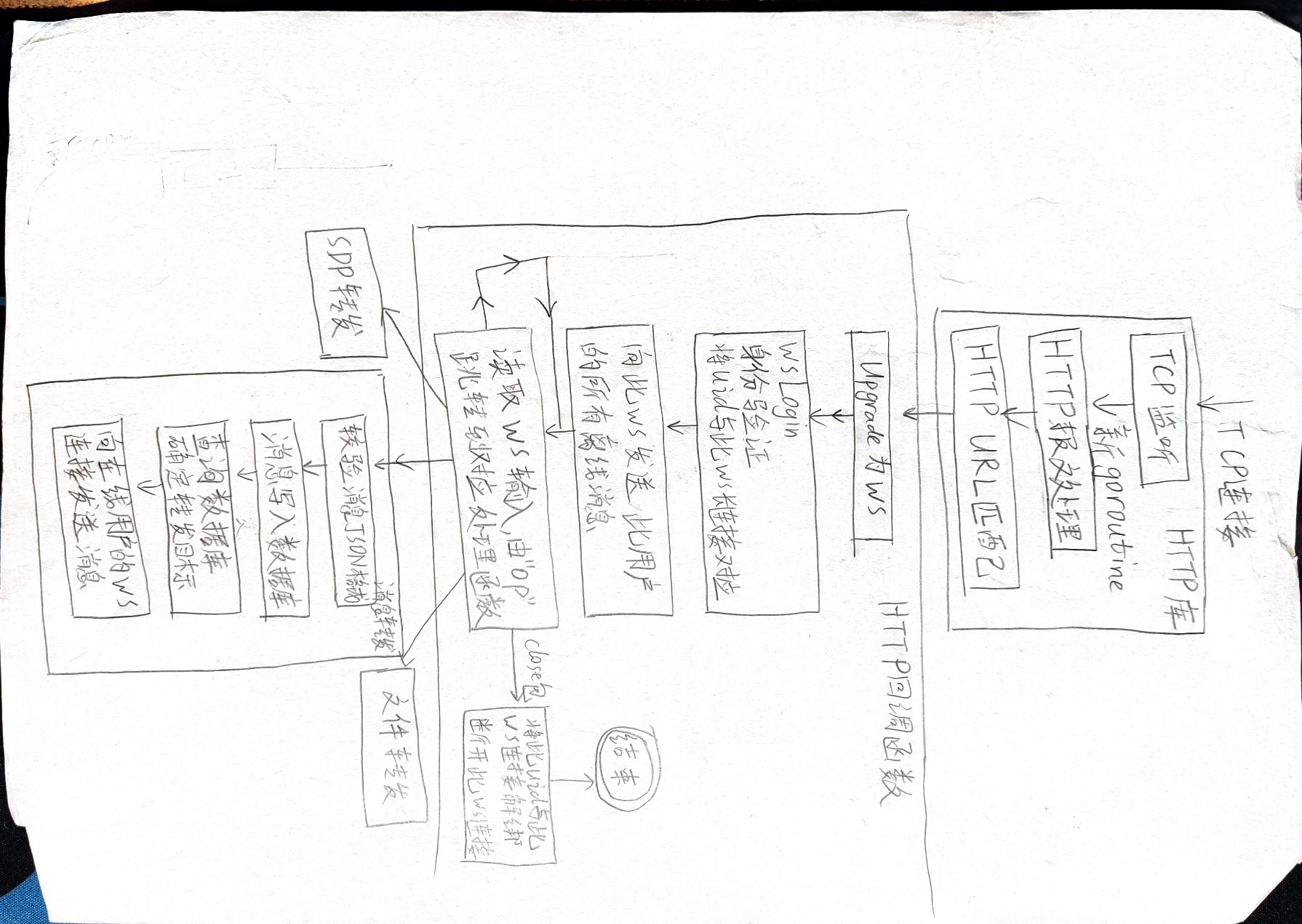
HTTP：

服务器需要同时处理HTTP和WS。对HTTP，我使用gin框架，架构如下：

（图片较复杂，在文档中难以展示，原图将一同提交）

总体而言，Gin框架负责监听TCP端口、为每个TCP连接新开goroutine处理、解析HTTP报文、根据路由来调用程序员编写的回调函数、向回调函数中传入“HTTP上下文”，程序员可以在回调函数中对此上下文进行读、写操作，也就完成了服务器对HTTP请求报文的读取和响应报文的发送。

而程序员编写的回调函数则涵盖了业务逻辑，包括请求字段的校验、用户登录状态的产生和验证、对象的生成、数据库操作、响应报文内容的填充等一系列操作。

WS：

对WS协议，我使用“gorilla/websocket”库。这个库需要先由go自带的HTTP库处理TCP连接，其操作与gin类似；然后在对应的HTTP回调函数中，先进行Upgrade操作，将TCP连接转换为一条WS连接，然后允许goroutine读写这条连接：同时最多一个goroutine读，同时最多一个goroutine写。

于是，我使用现有的处理这条TCP连接的goroutine（称之为“读者goroutine”，每条WS连接对应一个），以死循环形式读取WS连接发来的包。若包为Close包，则关闭这个WS连接、退出这个死循环（也就结束了这个“读者goroutine”）；；若包为字符包，则校验它的负载是否为JSON。若是，则通过JSON中的“op”字段来跳转到不同的处理函数中（“op”字段相当于HTTP中的URL，区分不同功能），它们有的直接转发（如根据目标uid转发SDP）；有的先保存到数据库，再在数据库中查询转发的目标用户（如会话中的其他成员），再检验目标用户是否在线，如果在线就向目标用户对应的WS连接发送JSON。处理这个包结束后，由于“读者goroutine”中的读取死循环，它又开始等待此连接的下一次输入。。。

并发控制：

对于HTTP，由于其是“无状态”协议，两次访问之间没有关系（没有共享资源），理论上无需并发控制。但在实际使用中，使用者还是会加入一些“状态”，例如“登陆状态”，这就引入了共享资源，需要进行并发控制。

我在一个全局map中存放了SessionID到uid的映射关系，用于服务器对于用户“登陆状态”的记录。由于可能多个HTTP连接同时访问、修改这个全局变量，因此我给它加上了多读单写锁sync.RWMutex（由于对它的写入仅发生在登录，而其它所有HTTP操作都需要读取它，满足“多读单写”的情况，使用它能够获得性能提升）。在登录代码前后加“写锁”，它会阻塞所有想要获取“写锁”和“读锁”的goroutine；在其它HTTP操作进行身份验证时加“读锁”，它会阻塞想要获取“写锁”的goroutine。

对于WS，虽然它是长连接，但在我的设计以及“聊天”的需求下，它的行为仍然类似HTTP的“请求-回复”：服务器从一个WS连接接收到消息，处理后，会向若干个其它WS连接发送数据。也就是说，在没有任何一个WS连接发来数据的时候，服务器也就不会被激活，也就不会向任何一个WS连接发送数据。但与HTTP不同的是，这份“回复”是写往别人的WS连接去的。容易想到，当有多个WS连接对应的goroutine都想往同一个WS连接中写入数据时，这就违背了“gorilla/websocket”库的“一个读取，一个写入”的规则了。所以要对WS连接的写入加锁，使得任何时刻，只有最多一个goroutine能对某个WS连接进行写入。（每个WS连接只由一个专门的、对应的“读者goroutine”死循环式的读取，不会被其它goroutine读，无需并发控制）

首先，“向某个uid转发数据”，需要用此uid找到对应的ws连接，这需要一个全局的“uid->WS连接指针”的map映射。这个map会在ws建立连接后，登录时添加新元素对、在连接断开时删除元素对、在其余时刻查询元素对。同上，使用多读单写锁sync.RWMutex保护它；其次，为满足“一个goroutine写入”的规则，在每个WS连接指针“身旁”，都放置一个互斥锁。一旦有goroutine通过uid找到了这个WS连接，想要对它进行写入，就先申请这个互斥锁。通过将WS连接指针和对应的互斥锁放到同一个结构体中，来达到“身旁”的效果。两个结构体定义如下：

// 用uid找到ws链接

type wsRouter struct {

    m map[int]\*wsLink

    l sync.RWMutex

}

// ws链接

type wsLink struct {

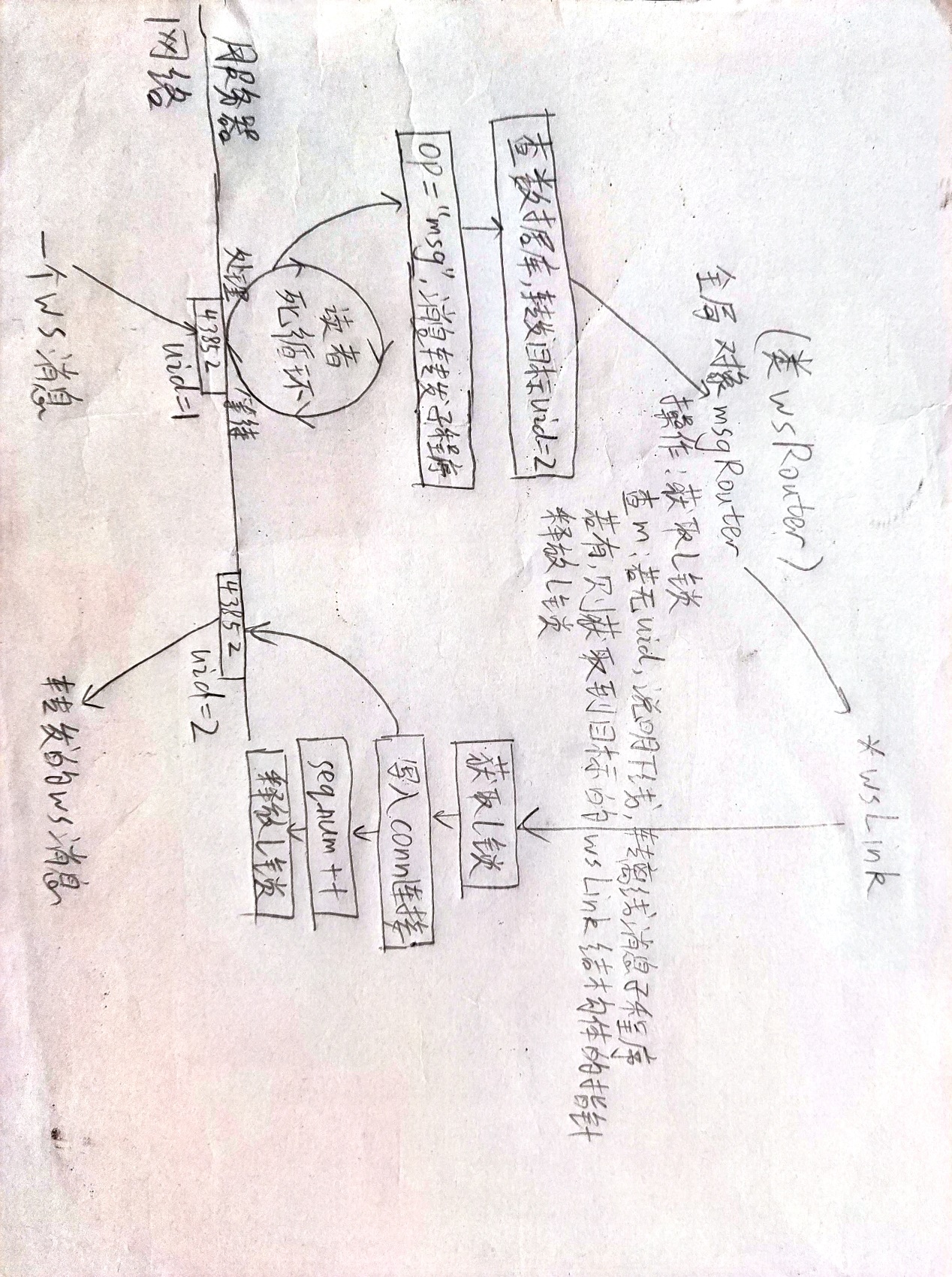
    conn    \*websocket.Conn

    seq\_num int // 发送时的序号，类似TCP

    l       sync.Mutex

}

一次WS消息转发的流程如下图所示：



错误处理：

Go语言内置了错误类型error。相当多的操作都有可能产生错误，如数据库连接时的用户密码错误、语句预编译时sql的语法错误、插入时主键重复的错误……这些错误不会导致程序core dump退出，而是会在相应的函数执行完毕后，返回一个非空的error，由调用者自行处理。作为一个服务器程序，其应该能提供7×24小时的服务，所以我选择将这些错误打印到日志中，而非终止程序。由于多个goroutine有可能并发打印日志，所以我使用了标准库“log”，它自带互斥锁，可以保证日志“不脏”“不乱”。

在Go中，对应其他语言的raise和catch的是Panic()和Recover()，而Gin框架默认对任何Panic()都进行Recover()，以确保服务器能够正常提供服务，随后它会将错误信息以及调用栈打印到日志中去。

数据库设计：

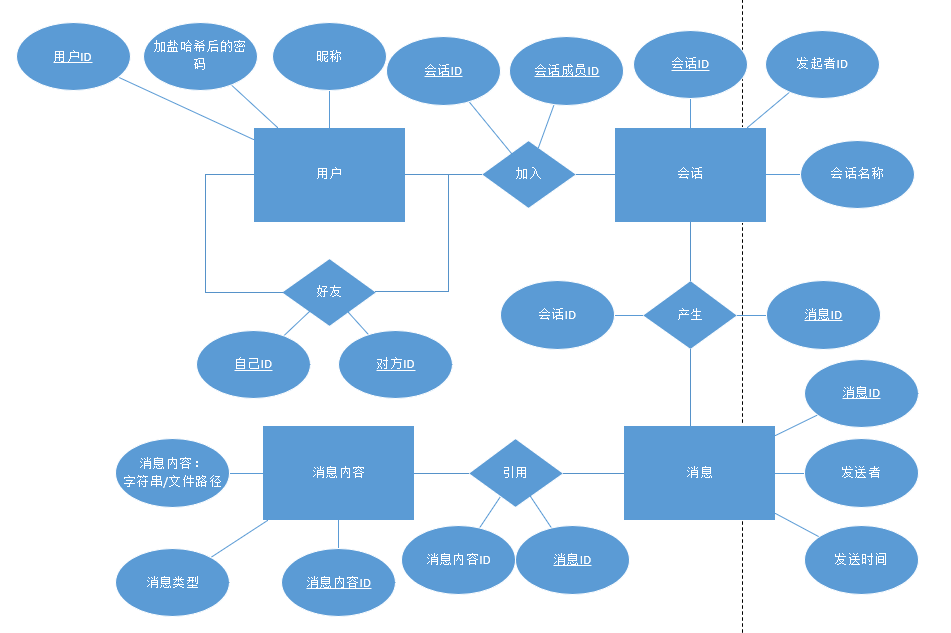
如果想要实现服务器验证登录、离线消息、离线文件中的任何一个功能，就都需要考虑服务器的数据存储问题。大致有三种解决办法：

1. 存储在服务器程序的变量中。优点是访问速度快、编写简单；缺点是一旦服务器进程重启，则数据全部丢失。

2. 存储在其它进程中，如Redis。优点是无需访问外存，速度较快，且服务器进程重启后，数据不会丢失；缺点是编写较为困难（使用进程间通信的几种方法来读写数据），且学校内没有系统学习过，学习成本较高，且服务器机器重启后会丢失所有数据，且存储空间有限。

3. 存储在数据库中，如MySQL。优点是技术成熟、学校教学也较为深入、数据是非电易失性的，且外存的空间较大；缺点是速度较慢。

最终我选择使用MySQL数据库，既是比较好地满足了需求，也能对数据库课程的知识进行练习。

 设计稿ER图如下：

用户表存储每个用户实体，包括用户编号、密码、昵称；

好友是用户之间的一对一关系，好友表中存储两列用户编号；

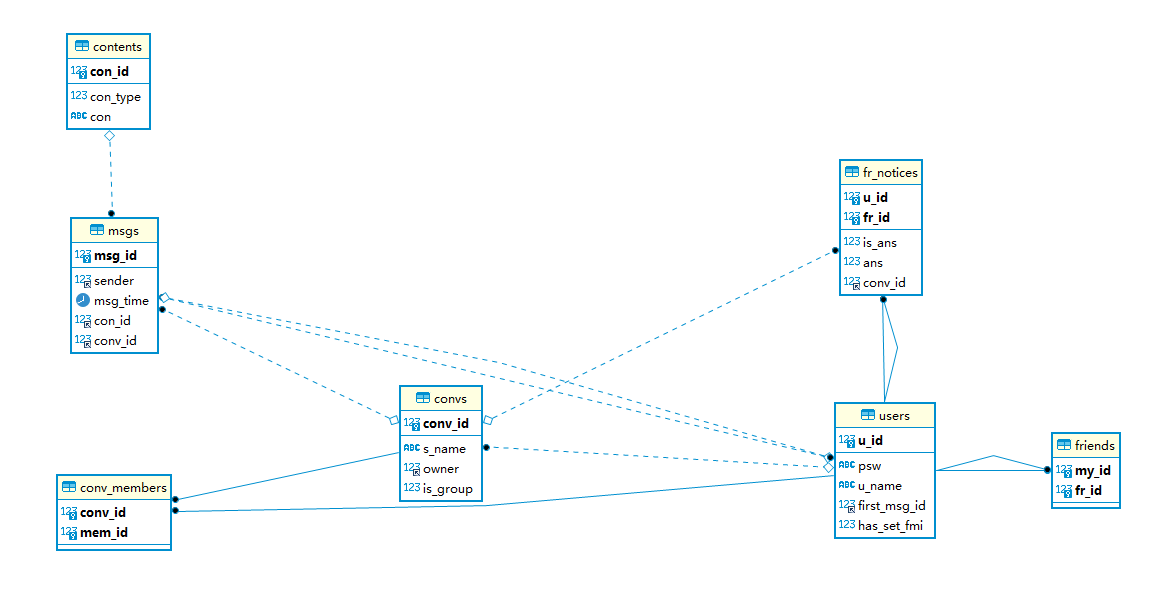
会话是消息的转发域，会话既可以是两两好友之间的会话，也可以是多个用户组成的群聊（群聊功能未实现）；若为群聊，则会话表中存储群聊的名称以及群主的用户编号；

会话成员是会话与用户之间的一对多关系，会话成员表中存储一列会话号和一列用户编号；

消息内容是服务器接收到的消息的具体内容，包括消息内容编号、消息内容类型（目前仅实现文本类型）、消息内容；

消息是消息内容、用户（发送者）、会话（消息内容所属的会话）之间的三元关系。将消息与消息内容分离，主要是为“转发功能”留余地，此时转发仅在消息表中添加一条记录，而消息内容表则没有增加。

实际数据库中的ER图如下：

 在实际编写中，除了以上字段，还为了离线消息、离线好友请求与回复等功能增加了若干表和表项。每当某用户有离线消息时，就在其对应的表中进行记录；当用户登录时，服务器查询这些表项，得到其“离线后收到的第一条消息的消息编号”，并将“此后所有该用户应收到的消息”全部推送给用户。

用Go调用MySQL，需要使用嵌入式SQL。为了避免SQL注入和同一SQL语句的重复解释，在服务器程序启动时，就将所有用到的SQL语句进行预编译。之后使用预编译完成的语句时，仅向数据库发送预编译SQL语句的编号和数据，减小网络开销；数据库也无需再对此语句进行解释、分析、优化等工作，直接执行准备好的目标代码即可。

**客户端：**

以electron作为前端框架。分为渲染进程和主进程，主进程负责本地文件处理和网络接口；渲染进程使用HTML/CSS/JS 做页面渲染。由于nodejs使用异步编程的思想，故不需要将网络通信单独作为一个进程（不会阻塞）。整体构架上主进程负责绝大部分逻辑处理和网络通信，通过\*\*进程间通信\*\*将信号和数据及时传输给渲染进程，渲染进程接收到数据后在经过简单的数据格式化处理和将其渲染在界面上。

1. 关键代码的描述
2. 关键代码1 客户端文件传输 毛弘卿

sendfile(filename) {

    console.log("start send file");

    let fileInfo = fs.statSync(filename);// 获取文件信息

    // 获取文件传输websocket

    let ws = this.filewebsocket;

    // 申请一片缓冲区，作为文件读取的缓冲单元

    var buf = new Buffer.alloc(BUFFSIZE);

    // 获取文件指针

    let fd = fs.openSync(filename);

    let readlength; // 读取的长度 单位:字节

    t.start = true;

    // 发送包含文件信息的数据包

    ws.send(

        JSON.stringify({

            op: "start",

            conv\_id: p.conv\_id,// 会话id

            name: filename,// 文件名称

            len: fileInfo.size,

            start: 0,

            end: Math.ceil(fileInfo.size / BUFFSIZE) - 1,

        })

    );

    do {

        // 将长度为buf.length的字节读入缓冲区中

        // 返回读取的字节数

        readlength = fs.readSync(

            fd,

            buf,

            0,

            buf.length,

            t.position

        );

        // 将缓冲区的数据通过websocket发送出去

        ws.send(buf.buffer);

        t.position += BUFFSIZE;

        if (t.pause) {

            // 进入等待死循环

        }

    } while (readlength === BUFFSIZE);

    // 关闭文件指针

    fs.closeSync(fd);

}

1. 关键代码2 客户端发送聊天信息 毛弘卿

sendmsg(msg) {

    let t = this.msgs[this.curname];

    // 视图更新

    t.msgList.push({

        time: Date().toString(),

        type: "text",

        content: t.input,

        pas: true,

    });

    // 发送数据包

    this.websocket.send(

        JSON.stringify({

            op: "msg",

            // 会话号

            conv\_id: t.conv\_id,

            // 发送消息的用户的uid

            sender: this.id,

            // 发送消息的时间

            time: Date().toString(),

            // 消息类型

            type: "text",

            // 消息内容

            content: t.input,

        })

    );

}

1. 关键代码3 客户端发起视频聊天 毛弘卿

// 发起聊天

// 新建p2p对象

this.peer = new SimplePeer({

    initiator: true,

    config: this.p2pconfig,

});

let peer = this.peer;

peer.on("signal", (data) => {

    // 将网络信息发送给对方

    this.sendbyWs({

        op: "connect",

        from: this.id,

        to: this.msgs[this.curname].frid,

        sdp: JSON.stringify(data),

    });

});

peer.on("connect", () => {

    // 在建立p2p连接后加入视频流

    peer.addStream(stream);

});

peer.on("stream", (newStream) => {

    // 接收视频流时，直接播放

    peervideo.srcObject = newStream;

    peervideo.play();

});

peer.on("close", () => {

    // 销毁视频流

    video.pause();

    video.srcObject

        .getTracks()

        .forEach((track) => track.stop());

    this.showvideo = false;

});

1. 关键代码4 客户端接受视频 毛弘卿

// 接受视频聊天

// 新建一个p2p处理对象

this.peer2 = new SimplePeer({

    config: this.p2pconfig,

});

let peer = this.peer2;

// 接受加载从发起方发送过来的sdp信息

peer.signal(JSON.parse(data));

peer.on("signal", (data) => {

    // 将网络信息发送给对方

    this.sendbyWs({

        op: "connect response",

        from: this.id,

        to: this.msgs[id].frid,

        sdp: JSON.stringify(data),

    });

});

peer.on("connect", () => {

    // 在建立p2p连接后加入视频流

    console.log("connected");

    peer.addStream(stream);

});

peer.on("stream", (stream) => {

    // 接收视频流时，直接播放

    video.srcObject = stream;

    video.play();

});

peer.on("error", (e) => {

    // 错误处理

});

peer.on("close", () => {

    // 销毁视频流

    localvideo.pause();

    localvideo.srcObject

        .getTracks()

        .forEach((track) => track.stop());

});

1. 关键代码5 服务器主程序 李程浩

//服务器主程序

package main

import (

    "log"

    "github.com/gin-gonic/gin"

)

// 全局变量

var sess sID

var msgRouter wsRouter

var fileRouter wsRouter

// 初始化全局变量

func init() {

    log.SetPrefix("LOG: ")

    log.SetFlags(log.Ldate | log.Ltime | log.Lshortfile)

    sess.m = make(map[string]int)

    msgRouter.m = make(map[int]\*wsLink)

    fileRouter.m = make(map[int]\*wsLink)

}

func main() {

    // rand.Seed(1)

    r := gin.Default()

    r.POST("/login", login)

    r.POST("/addfriend", addfriend)

    r.POST("/resfriend", resfriend)

    r.POST("/name", nameService)

    r.POST("/friendlist", friendList)

    r.POST("/convlist", convList)

    r.POST("/convmemlist", convMemList)

    go wsStarter()

    err := r.Run(":43851")

    if err != nil {

        panic(err)

    }

}

1. 关键代码6 “登录”的HTTP部分 李程浩

// 登录

func login(c \*gin.Context) {

    var u db.User

    if err := c.ShouldBindJSON(&u); err != nil {

        c.JSON(400, gin.H{"res": "NO", "reason": "json bind error"})

        return

    }

    if u.Login() {

        sid := sess.set(u.Id)

        c.JSON(http.StatusOK, gin.H{"res": "OK", "sid": sid, "name": u.Name})

        log.Println("user", u.Id, "http login suss")

    } else {

        c.JSON(http.StatusOK, gin.H{"res": "NO", "reason": "wrong password"})

    }

}

1. 关键代码7 聊天消息转发 李程浩

func msgForward(uid int, b []byte) (err error) {

    // 包格式处理

    var pkg, no\_content struct {

        wsMain

        db.WsMsg

    }

    err = json.Unmarshal(b, &pkg)

    if err != nil {

        return

    }

    pkg.Sender = uid

    pkg.Time = time.Now().Format("2006-01-02 15:04:05")

    no\_content = pkg

    no\_content.Content = ""

    // 寻找目标

    mems, err := db.GetOtherConvMems(uid, pkg.Conv\_id)

    if err != nil {

        return

    }

    if len(mems) == 0 {

        log.Println("a msg with no forward target:", string(b))

        return

    }

    // 保存消息

    msg\_id, suss := pkg.Save()

    if !suss {

        log.Println(no\_content, "save fail!")

    }

    // 转发

    var u db.User

    for \_, i := range mems {

        msgRouter.l.RLock()

        link, ok := msgRouter.m[i]

        msgRouter.l.RUnlock()

        // 目标不在线, 设置离线消息提醒

        if !ok {

            u.Id = i

            err = u.StoreOfflineMsg(msg\_id)

            if err != nil {

                log.Println(err)

            }

            log.Println("save notice for user:", u.Id, ", of saved msg:", msg\_id)

            continue

        }

        link.l.Lock()

        pkg.Seq = link.seq\_num

        err := link.conn.WriteJSON(pkg)

        link.seq\_num++

        link.l.Unlock()

        if err != nil {

            log.Println(err)

            continue

        }

        log.Println("msg forward:", no\_content, "to", i)

    }

    return nil

}

1. 测试及结果分析

由于功能较多、使用文字图片描述较为困难。我和其它同学使用软件，录制了三段视频放在压缩包下，分别展示聊天消息转发、（断点续传）文件传输和p2p视频功能。各功能均测试成功。

1. 实验结论

经过多个子系统的综合设计和实现，我们完成了一个功能较为丰富，具有一定使用价值的聊天软件。

1. 总结及心得体会

1. 这是我（李程浩）第一次与别人合作进行编程。由于一开始就基本商定好了我们软件的功能，以及我们两人之间的接口：网络协议文档，整体的合作较为顺利。

2. 在之前的操作系统实验中，我（李程浩）的[文件系统](https://github.com/Live4dreamCH/MyExt2)编写过程中没有进行单元测试，导致全部完成后，难以找出仍旧存在的bug；在之后的[组成实验](https://github.com/Live4dreamCH/Computer-Organization-Experiment)和本次[网络实验](https://github.com/Live4dreamCH/socket_backend)中，我重视起单元测试，有效地将bug扼杀于萌芽之中，整体代码的稳定性提升了。

3. 我们的进度控制较差，将较多的任务堆积到了最后。有可能造成项目的失败，以及质量的下降。这几周学习软件工程后，逐渐明白了进度控制的一些常用手段，如里程碑、甘特图等，将抓住之后的机会进行实践。

4. 端到端地实操了网络编程。由于我们之前都尝试过通过socket，在两端调用传输层协议TCP、UDP等，传输数据，但将其应用于实践还是第一次。

附件

1.源码文件 在压缩包下，我们也开源了我们的代码：

服务器、数据库代码 <https://github.com/Live4dreamCH/socket_backend>

客户端代码 <https://github.com/mhq1065/netChat>

2.参考资料（仅服务器端）

Go指南：基础语法

<https://tour.go-zh.org/welcome/1>

Go Web编程

<https://github.com/astaxie/build-web-application-with-golang/blob/master/zh/preface.md>

Gin框架官方文档

<https://github.com/gin-gonic/gin#quick-start>

Gorilla WebSocket库官方文档

<https://pkg.go.dev/github.com/gorilla/websocket>

Go database/sql库的用法、文档

<https://segmentfault.com/a/1190000003036452>

<http://cngolib.com/database-sql.html>