CS305 Project An Online Conferencing System

Group Members: 11911808 刘仁杰，11912640 林慧燕，11912922 彭琳凯

1. Background and related network protocol survey

Background

Online conference system refers to a system in which many people meet face to face on the network at the same time. A system of equipment used by two or more individuals or groups in different places to transmit sound, video and documents to each other through transmission lines and multimedia equipment to achieve real-time and interactive communication for meeting purposes.

Related network protocol survey

Session Initiation Protocol (SIP) is an application-layer signaling and control Protocol. A session for creating, modifying, and releasing one or more participants. These sessions can be Internet multimedia meetings, IP calls, or multimedia distribution. Participants in a session can communicate via multicast, unicast, or a mixture of the two. SIP interoperates with the Resource Reservation Protocol (RSVP), which is responsible for voice quality. It also collaborates with several other protocols, including lightweight Directory Access Protocol (LDAP) for location, Remote Authenticated Dial-in User Service (RADIUS) for authentication, and RTP for real-time transport.

The RTP protocol specifies the standard packet format for delivering audio and video over the Internet. It was originally designed as a multicast protocol, but has been used in many unicast applications. RTP is commonly used in streaming media systems (in conjunction with RTSP), video conferencing, and Push to Talk systems (in conjunction with H.323 or SIP), making it the technical foundation of the IP telephony industry. The RTP protocol is used together with the RTP control protocol RTCP and is built on top of UDP.

2. Overall system structure and system protocol description

Overall system structure

Server

Upon the initialization of the server, four ports including main connection, video transmission, audio transmission and screen transmission will start listening to tcp socket connection. Among these four ports, the main port listen on the connection request of a new client and relegate the client to a new server socket while the other three ports receive corresponding connection request and start a new thread to receive data seperately.

Client

We created six sockets for connecting to the server, transmitting and receiving audio, video, screen, listening to remote control signals, and sending control signals. Each client has a Qt interface that handles window events. There is a room_id recording current meeting room, which is also responsible for controlling the behavior of video, audio and screen trasmission action.

System protocol description

Meeting

Every meeting retains a main tcp socket between server and client, client actions include `create room`, `join room`, `quit room`, `close room`, `set admin`, `transfer host`.

We define our own message control protocol based on tcp socket. Like `http` packet, our packet also consists of `header` and `data`, which are sperated by `\r\n\r\n`. Moreover, the tcp socket is set non-blocking, thus we directly receive the packet sent from remote and tcp protocol guarantees the reliablity and order of packets, which means that there is no need for us to state the length of the packet and the sequnce id in the header.

So our packet would be sent and parsed as follows:

```
def send_data(sock, header, data):
    pack = header + b'\r\n\r\n' + data
    sock.sendall(pack)

def parse_data(raw_data):
    raw_data = raw_data.decode().split('\r\n\r\n')
    header = raw_data[0]
    data = raw_data[1]
    return header, data
```

Audio transmission

Every audio receiving and sharing action attains an exclusive tcp socket connected to the server, which parse and packet data in a specific protocol format. Since tcp guarantee the order of its packets, we just still don't need to care about this stuff.

Moreover, since there is only one output stream of audio data, there is also no need for us to distinguish the audio data between different clients, we just need to consider the completeness of one audio packet since it is too large to fit in one receiver buffer.

Under this circumstance, our audio transfer protocol simply consists of two parts: `data stream` composed of the bytes stream length of the audio stream appended by a flow of audio data stream.

Accordingly, the sending format is shown below:

```
sock.sendall(struct.pack("L", len(senddata)) + senddata)
```

And the receiving rule is described as follows:

```
data = "".encode("utf-8")
payload_size = struct.calcsize("L")
while len(data) < payload_size:
    data += sock.recv(81920)
packed_size = data[:payload_size]
data = data[payload_size:]
msg_size = struct.unpack("L", packed_size)[0]
while len(data) < msg_size:
    data += sock.recv(81920)
frame_data = data[:msg_size]
data = data[msg_size:]
```

Video transmission

Every video receiving and sharing action attains a exclusive tcp socket connected to the server, which parse and packet data in a specific protocol format. Since tcp guarantee the order of its packets, we just still don't need to take concerns on this stuff.

Howerver, we need to distinguish the video data packets berween different clients since for each client, there will be a unique window to display their video content.

Under this circumstance, our video transfer protocol simply consists of two parts: `ip information` and `data stream`. `ip information` is made up of the bytes stream length of the ip appended by the sender ip and `data stream` is composed of the bytes stream length of the video stream appended by a flow of video data stream.

Accordingly, the sending format is shown below:

```
sock.sendall(struct.pack("L", len(ip_b)) + ip_b + struct.pack("L", len(zdata)) +
zdata)
```

And the receiving rule is described as follows:

```
payload_size = struct.calcsize("L")
while len(data) < payload_size:
    data += sock.recv(81920)
ip_size = struct.unpack("L", data[:payload_size])[0]
data = data[payload_size:]
while len(data) < ip_size:
    data += sock.recv(81920)
ip = data[:ip_size].decode()
data = data[ip_size:]
while len(data) < payload_size:
    data += sock.recv(81920)
msg_size = struct.unpack("L", data[:payload_size])[0]
data = data[payload_size:]
```

Remote desktop control

Remote desktop control is an end-to-end implementation, so we don't need to design a data transfer header. Firstly, the client that is controlled needs to continuously transmit desktop screenshots to the controlling client. Here, we will send them twice. The first time the length of picture data is sent, and the second time the picture data is sent. As shown below:

```
lenb = struct.pack(">BI", 1, len(self.imbyt))
conn.sendall(lenb)
conn.sendall(self.imbyt)
```

The controller receives the data and displays it in the CV2 window. The CV2 window will listen for mouse and keyboard events and then send them to the controlled end in the following form:

```
self.sock.sendall(struct.pack('>BBHH', 1, 100, x, y))
```

1 represents the left mouse button, 100 represents the press, x and y represents the coordinates of the mouse. In this way, the controlled end will make the corresponding mouse and keyboard actions after receiving the data.

System mechanism description

Meeting control mechanism

- A client can create a meeting by sending a messge beginning with `create room\r\n\r\n`.
- The client creating a meeting will be the room host and have the privilege to end the whole meeting.
- Other clients can join the meeting by inputting the meeting id and the backend will send `join room\r\n\r\n` to the server.
- Clients joining the meeting is regarded as normal client with no privilege to end the whole meeting.
- Meeting host can assign other clients to be the administrator, which will have the privilege to end the meeting.
- Host can transfer the host privilege to another client, then the original host will be a normal client.
- There are four button for clients attending a meeting controling the on and off of the audio, video, screen and control behavior.
- The meeting host and administrators can end the whole meeting and then other clients will automatically be quitted from the same meeting.

Transfer control mechanism

- One important mechanism for the data transfer control is the server's data receiving and forwarding mechanism.
- Since each audio, video and screen data sending goes in a seperated socket with multi-thread concurrently, the server should not forward a piece of data stream upon receiving since a complete packet might consists of multiple pieces.
- Thus, we need to retain a queue for different data stream, namely audio, video and screen. For each data receiving socket, we continously receive data flow from the backend buffer until we get a complete packet and then push it into the queue.
- Meanwhile, each meeting will sequentially and periodically broadcast the packet at the head of the queue to all the client socket in the same meeting via multi-thread.
- Therefore, the client receiving end will get sequential and complete data stream of video, audio and screen packets.

Remote control mechanism

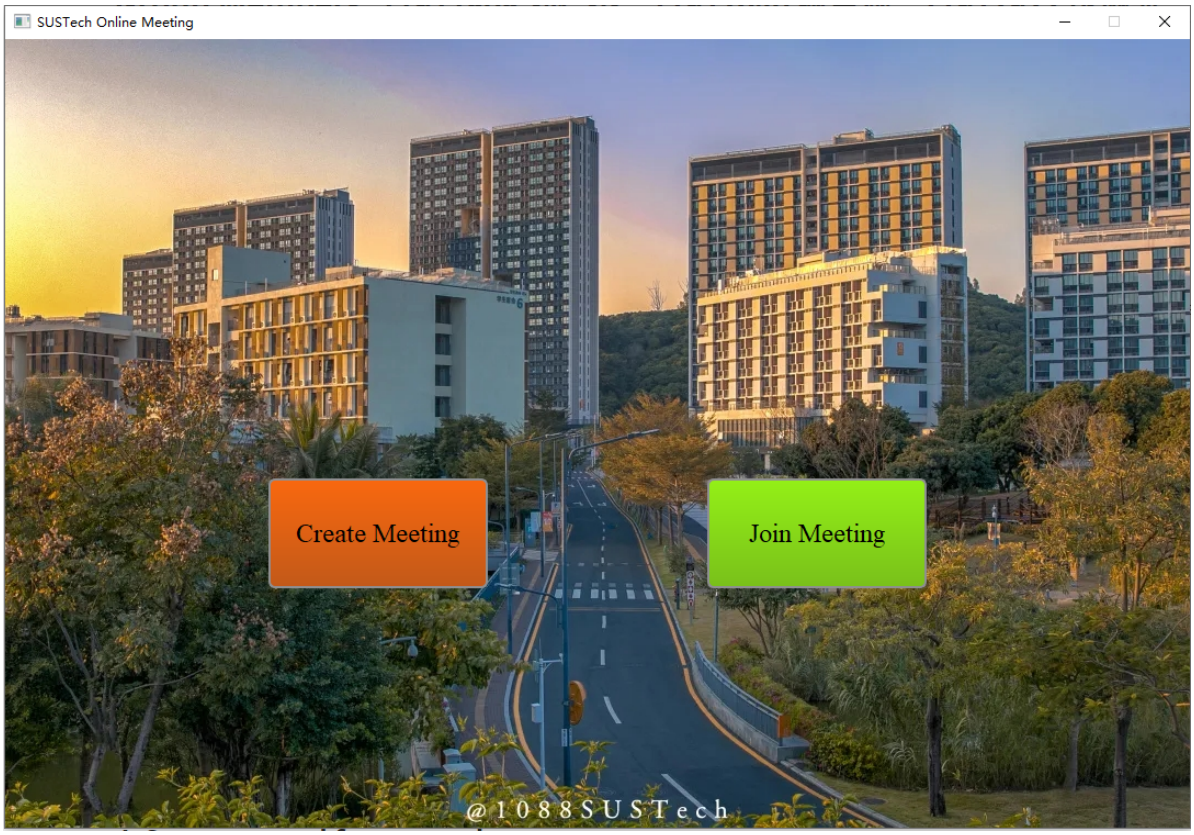
- Participants can select the client that they want to control from the list.
- When a control request is received by the controlled client, a popup is generated asking whether or not to accept the request.
- If yes, the control succeeds.

- If no, the controller will receive a denying message.

3. System testing results and packet capture analysis

System testing results

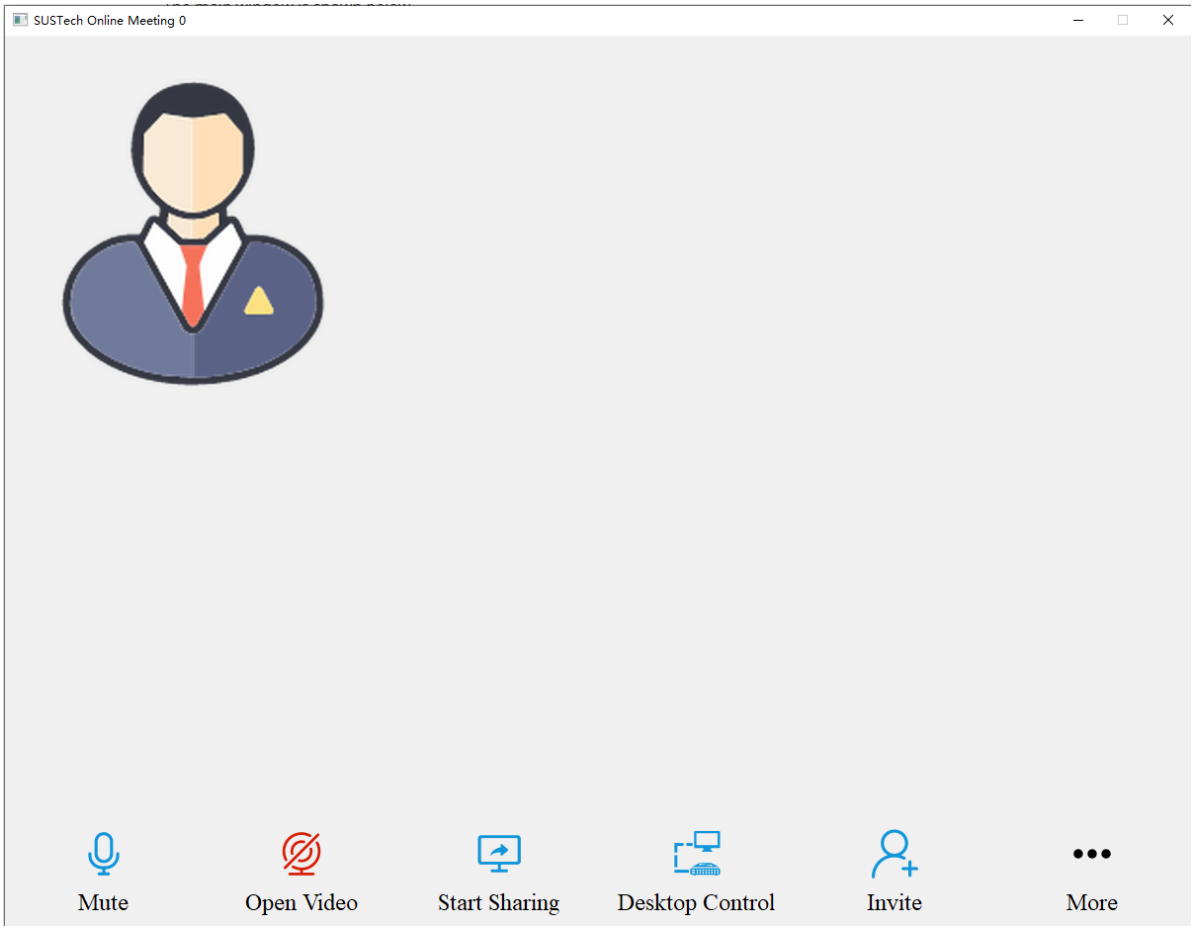
Main Window:



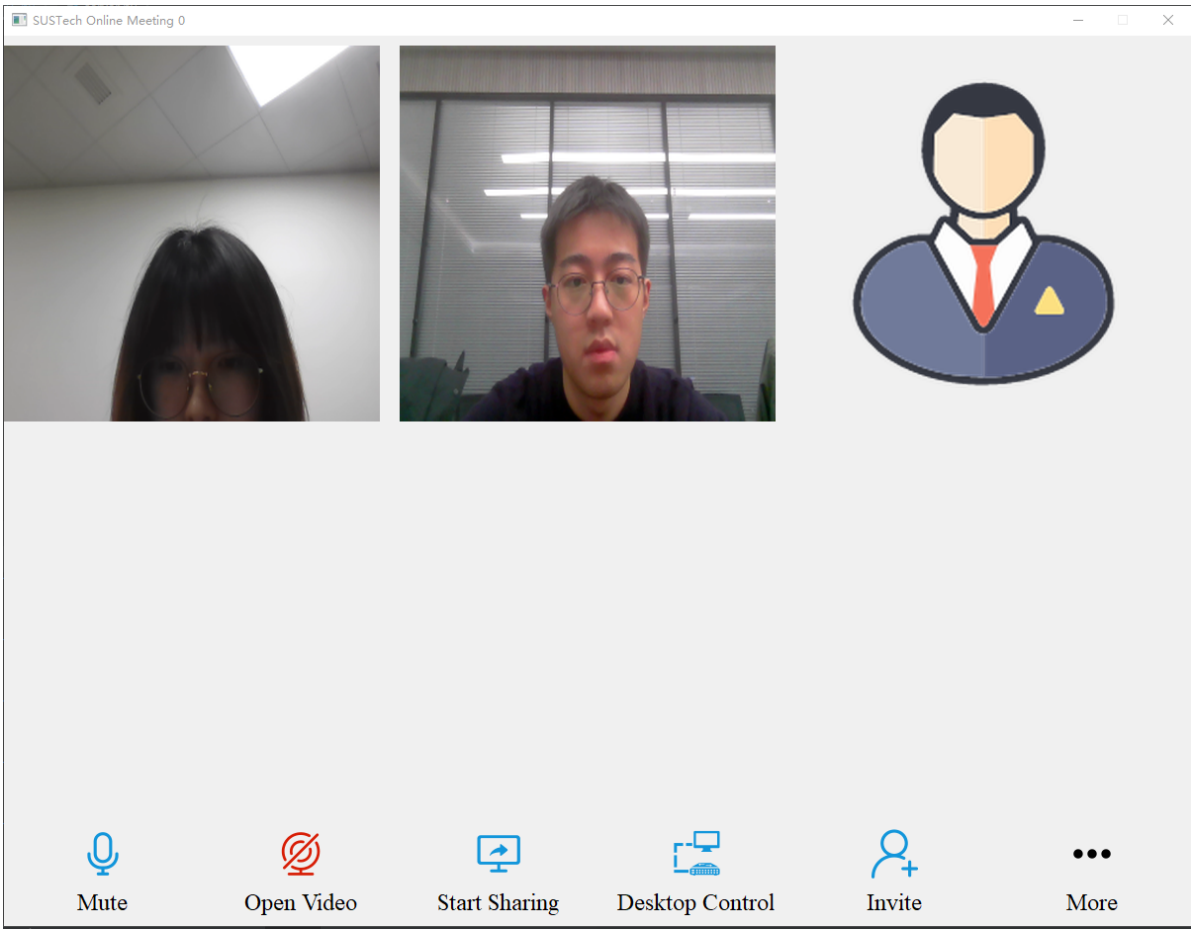
When joining a meeting, a user will be prompted to input a meeting id:

If the meeting id is incorrect, a error message will appear:

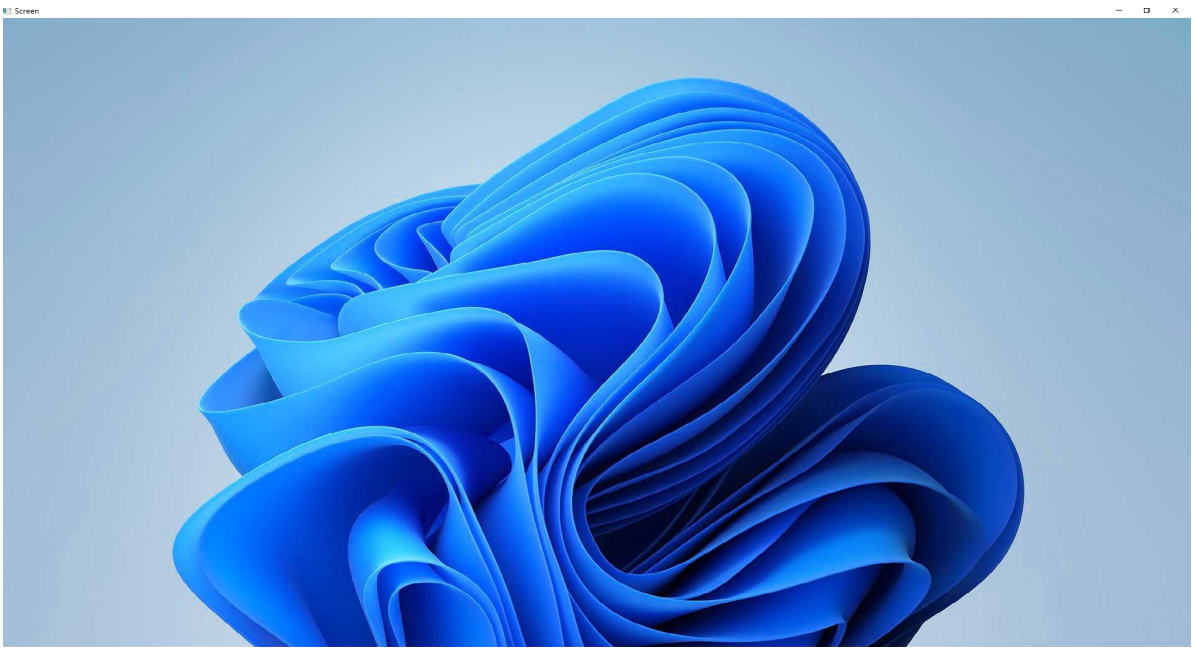
If the meeting id is correct or the user create a meeting, the meeting window will appear:



Real-time video transmission.

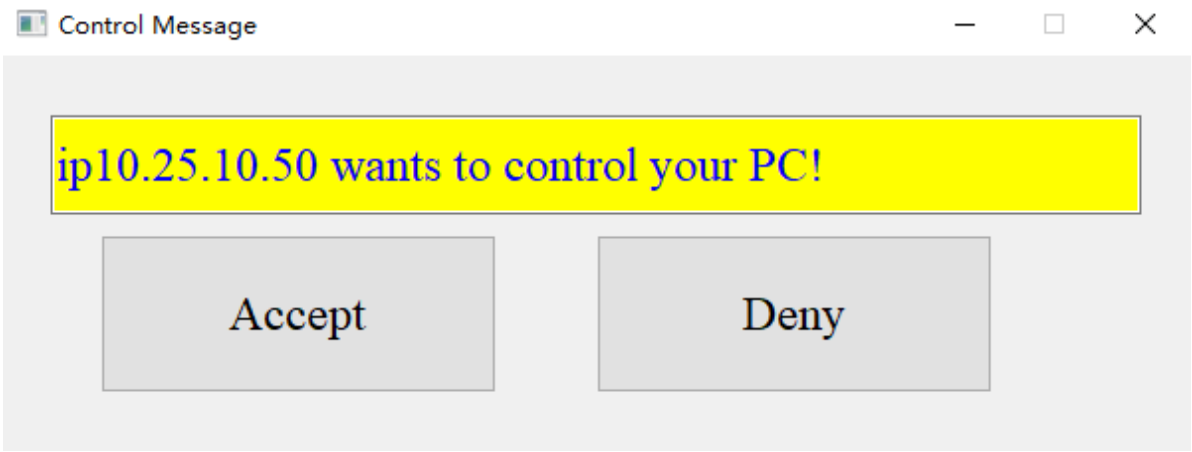


Screen sharing

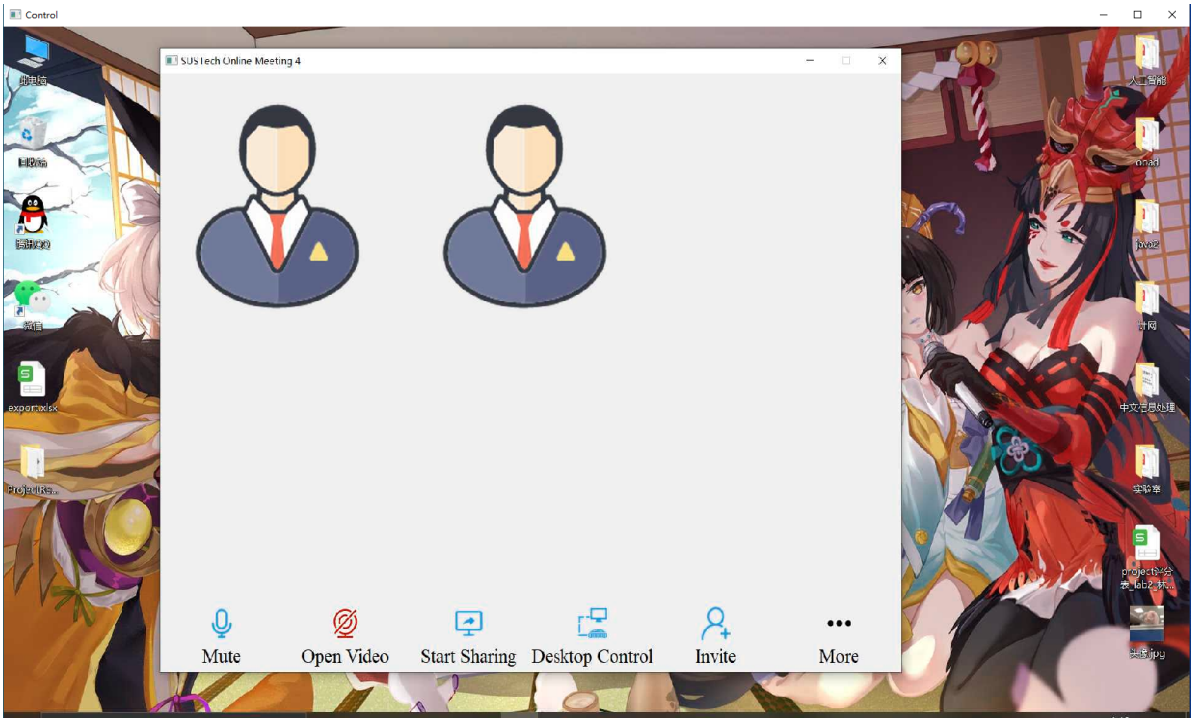


Screen Control:

When one want to control others' desktop, a message will pop up.

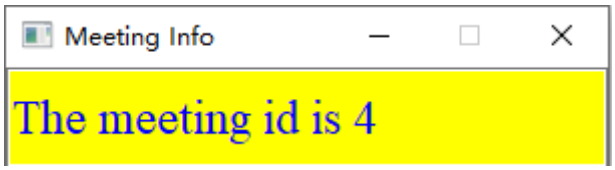


When the user accept the request, a control window will be shown.

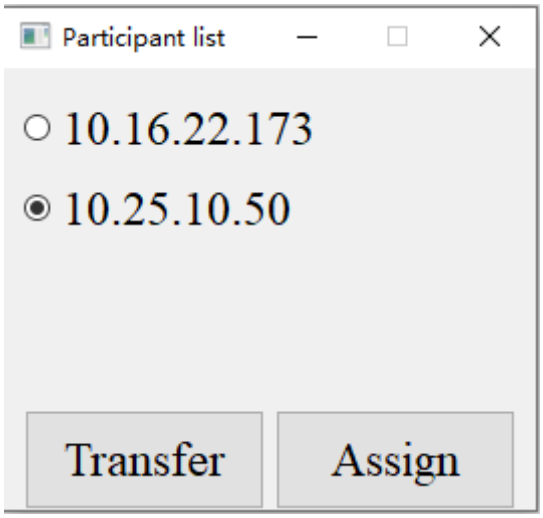


If the user deny the request, a pop up window will appear, e.g.

When the invite button is pressed, a message about meeting information will be shown.



When the more button is pressed, the user can see all participants and host can transfer itself and assign administrators.

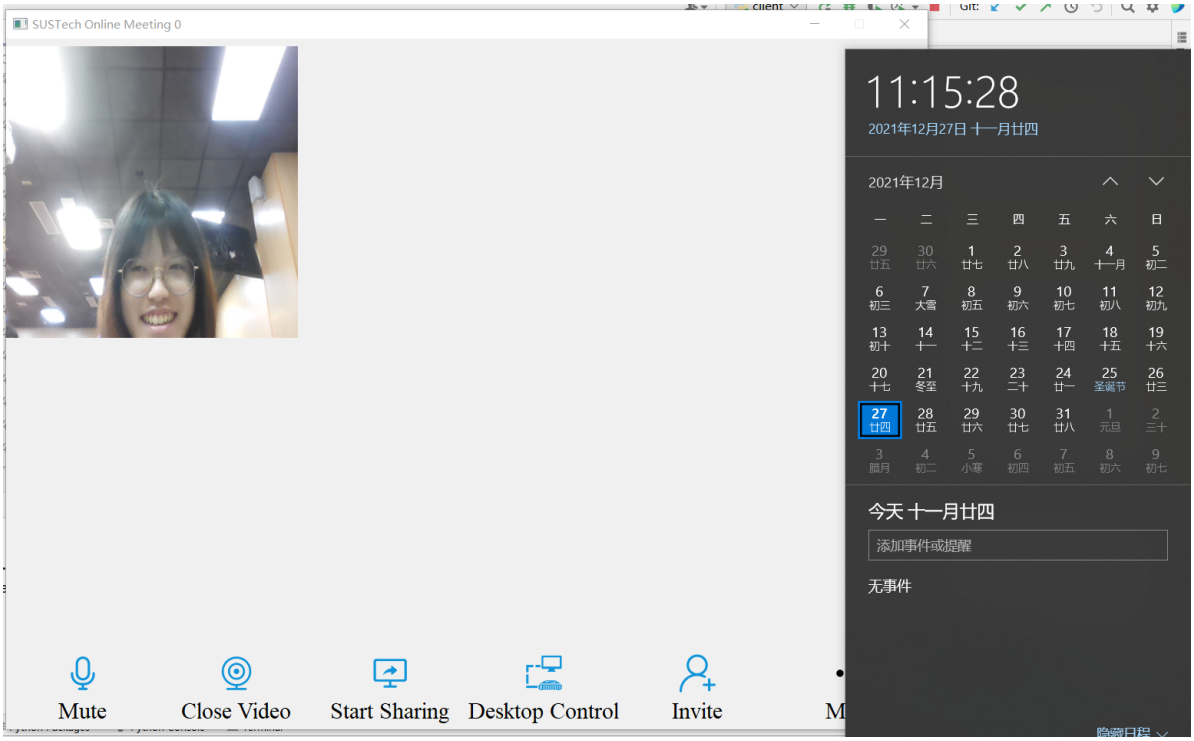


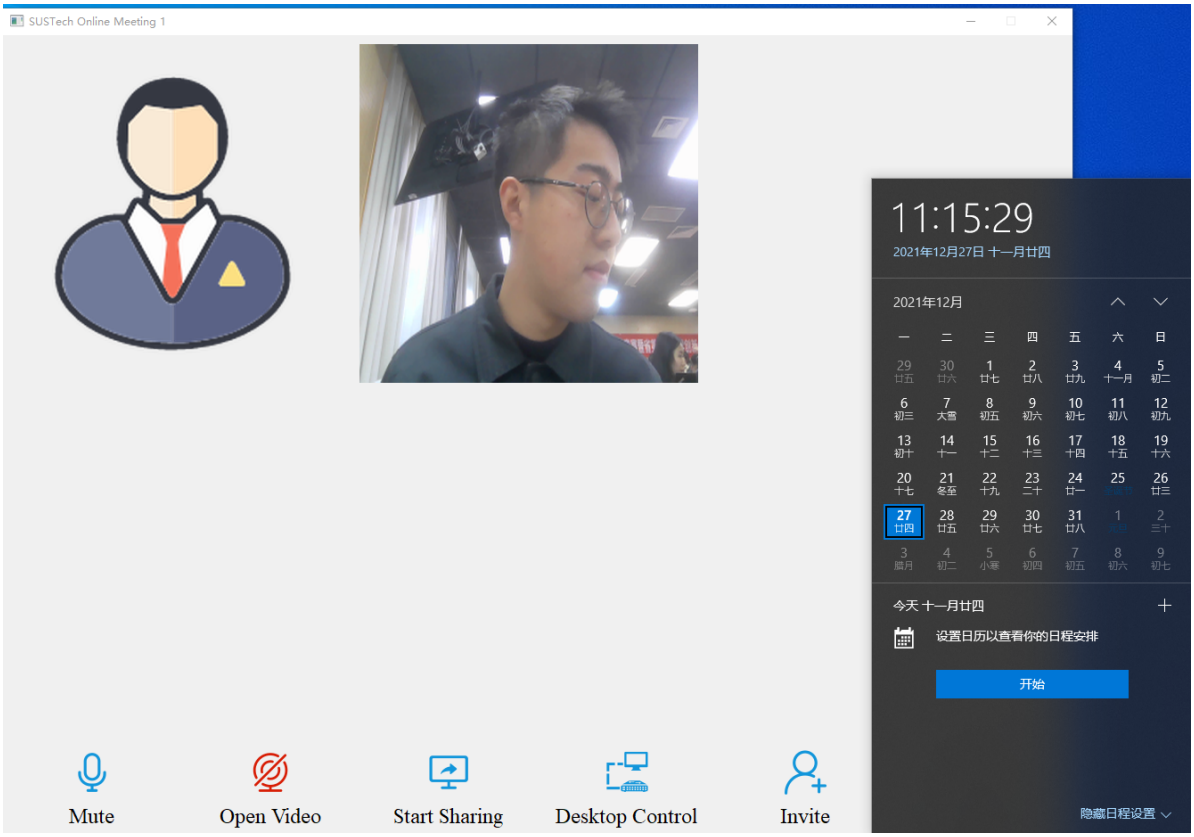
For host and administrators, when they want to close the meeting window, they will see

For others, they can only see

When the meeting is closed, they will return to the main window.

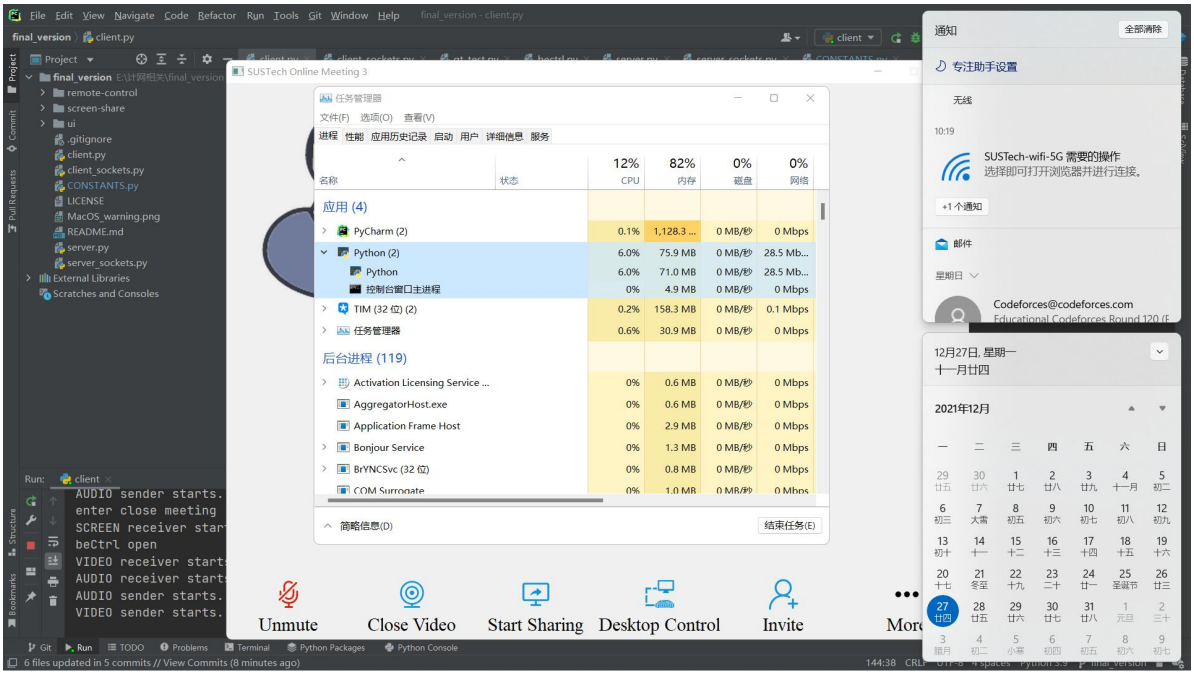
Multiple meeting at the same time is also supported.

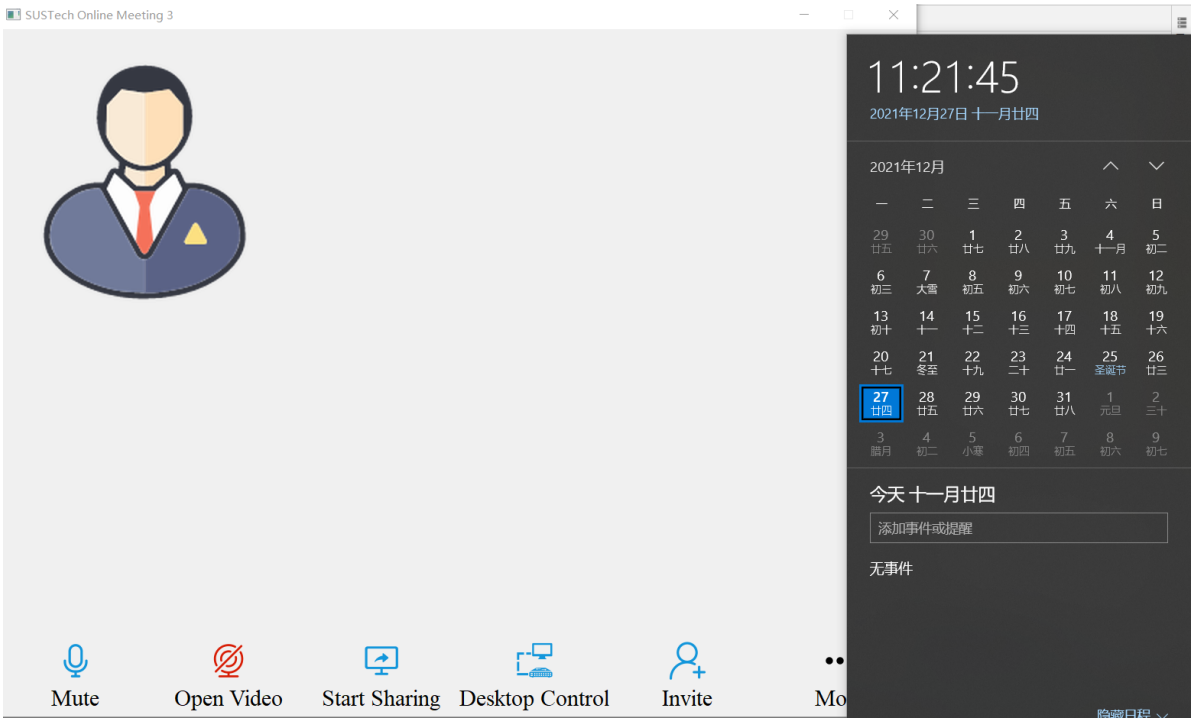




Exception handling:

When one client force to stop the program, the meeting is still working.





Packet capture analysis

Example: client1 ip is 10.25.10.50, client2 ip is 10.26.139.226 server ip is 10.26.22.173. The port for main connection is 5000. Ports for video, audio, screen sharing and desktop control are 5001, 5002, 5003 and 5004 respectively.

Basic connection established:

6160	53.150293	10.25.10.50	10.16.22.173	TCP	66 60815 → 5000 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
6161	53.150373	10.16.22.173	10.25.10.50	TCP	66 5000 → 60815 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
6162	53.154477	10.25.10.50	10.16.22.173	TCP	60 60815 → 5000 [ACK] Seq=1 Ack=1 Win=131328 Len=0

Client1 pressed the button create meeting, sockets for different usage are connected with different ports:

```
36613 315.109753  10.25.10.50    10.16.22.173  TCP   70 60815 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=16
36623 315.144953  10.16.22.173   10.25.10.50   TCP   72 5000 → 60815 [PSH, ACK] Seq=1 Ack=17 Win=262656 Len=18
36624 315.149074  10.25.10.50    10.16.22.173  TCP   65 60815 → 5000 [PSH, ACK] Seq=17 Ack=19 Win=131328 Len=11
36626 315.152308  10.25.10.50    10.16.22.173  TCP   66 59909 → 5001 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
36627 315.152378  10.16.22.173   10.25.10.50   TCP   66 5001 → 59909 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
36628 315.152429  10.25.10.50    10.16.22.173  TCP   66 59910 → 5002 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
36629 315.152480  10.16.22.173   10.25.10.50   TCP   66 5002 → 59910 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
36630 315.152507  10.25.10.50    10.16.22.173  TCP   66 59911 → 5002 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
36631 315.152530  10.16.22.173   10.25.10.50   TCP   66 5002 → 59911 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
36632 315.152557  10.25.10.50    10.16.22.173  TCP   66 59912 → 5003 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
36633 315.152602  10.16.22.173   10.25.10.50   TCP   66 5003 → 59912 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
36634 315.157876  10.25.10.50    10.16.22.173  TCP   60 59909 → 5001 [ACK] Seq=1 Ack=1 Win=131328 Len=0
36635 315.157876  10.25.10.50    10.16.22.173  TCP   60 59911 → 5002 [ACK] Seq=1 Ack=1 Win=131328 Len=0
36636 315.157882  10.25.10.50    10.16.22.173  TCP   60 59910 → 5002 [ACK] Seq=1 Ack=1 Win=131328 Len=0
36637 315.157938  10.25.10.50    10.16.22.173  TCP   60 59912 → 5003 [ACK] Seq=1 Ack=1 Win=131328 Len=0
36639 315.164704  10.25.10.50    10.16.22.173  TCP   71 59911 → 5002 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=17
36640 315.164704  10.25.10.50    10.16.22.173  TCP   73 59909 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=19
36641 315.164705  10.25.10.50    10.16.22.173  TCP   73 59912 → 5003 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=19
36642 315.164715  10.25.10.50    10.16.22.173  TCP   73 59910 → 5002 [PSH, ACK] Seq=1 Ack=1 Win=131328 Len=19
36646 315.168985  10.16.22.173   10.25.10.50   TCP   79 5000 → 60815 [PSH, ACK] Seq=19 Ack=28 Win=262656 Len=25
```

Client1 opened audio, video and screen sharing, the transmission started:

```
10.25.10.50    10.16.22.173   TCP   1514 52910 → 5002 [ACK] Seq=9462737 Ack=12 Win=131328 Len=1460
10.25.10.50    10.16.22.173   TCP   1514 52910 → 5002 [ACK] Seq=9464197 Ack=12 Win=131328 Len=1460
10.16.22.173   10.25.10.50    TCP     54 5002 → 52910 [ACK] Seq=12 Ack=9465657 Win=262656 Len=0
10.25.10.50    10.16.22.173   TCP   1514 52910 → 5002 [ACK] Seq=9465657 Ack=12 Win=131328 Len=1460

10.25.10.50    10.16.22.173   TCP   1514 52911 → 5001 [ACK] Seq=17471030 Ack=12 Win=131328 Len=1460
10.25.10.50    10.16.22.173   TCP   1514 52911 → 5001 [ACK] Seq=17472490 Ack=12 Win=131328 Len=1460
10.16.22.173   10.25.10.50    TCP     54 5001 → 52911 [ACK] Seq=12 Ack=17473950 Win=262656 Len=0
10.25.10.50    10.16.22.173   TCP   1514 52911 → 5001 [ACK] Seq=17473950 Ack=12 Win=131328 Len=1460
10.25.10.50    10.16.22.173   TCP   1514 52911 → 5001 [ACK] Seq=17475410 Ack=12 Win=131328 Len=1460

10.25.10.50    10.16.22.173   TCP   1514 52922 → 5003 [ACK] Seq=21895842 Ack=12 Win=131328 Len=1460
10.25.10.50    10.16.22.173   TCP   1514 52922 → 5003 [ACK] Seq=21897302 Ack=12 Win=131328 Len=1460
10.16.22.173   10.25.10.50    TCP     54 5003 → 52922 [ACK] Seq=12 Ack=21898762 Win=262656 Len=0
10.25.10.50    10.16.22.173   TCP   1514 52922 → 5003 [ACK] Seq=21898762 Ack=12 Win=131328 Len=1460
```

Client1 closed audio, video and screen sharing, the transmission stopped:

```
10.25.10.50    10.16.22.173   TCP     60 52910 → 5002 [FIN, ACK] Seq=26109831 Ack=12 Win=131328 Len=0
10.16.22.173   10.25.10.50    TCP     54 5002 → 52910 [ACK] Seq=12 Ack=26109832 Win=262656 Len=0

10.25.10.50    10.16.22.173   TCP     60 52911 → 5001 [FIN, ACK] Seq=150831870 Ack=12 Win=131328 Len=0
10.16.22.173   10.25.10.50    TCP     54 5001 → 52911 [ACK] Seq=12 Ack=150831871 Win=262656 Len=0
10.25.10.50    10.16.22.173   TCP     60 59909 → 5001 [ACK] Seq=20 Ack=271469632 Win=635904 Len=0
10.16.22.173   10.25.10.50    TCP     73 5001 → 59909 [PSH, ACK] Seq=271469632 Ack=20 Win=262656 Len=19
10.25.10.50    10.16.22.173   TCP     60 59909 → 5001 [ACK] Seq=20 Ack=271469651 Win=635904 Len=0

10.25.10.50    10.16.22.173   TCP     60 50751 → 5003 [FIN, ACK] Seq=482232 Ack=12 Win=131328 Len=0
10.16.22.173   10.25.10.50    TCP     54 5003 → 50751 [ACK] Seq=12 Ack=482233 Win=262656 Len=0
```

When client2 joined the meeting, the server will broadcast related information:

```
 85973 550.354412  10.16.22.173   10.25.10.50   TCP   97 5000 → 60815 [PSH, ACK] Seq=68 Ack=1 Win=1026 Len=43
 86011 550.572631  10.25.10.50    10.16.22.173  TCP   60 60815 → 5000 [ACK] Seq=1 Ack=111 Win=512 Len=0

Frame 85973: 97 bytes on wire (776 bits), 97 bytes captured (776 bits) on interface \Device\NPF_{3E6FB536-06A6-4678-AB75-3F7EF060ADCA}, id 0
Ethernet II, Src: Giga-Byt_3c:35:cb (18:c0:4d:3c:35:cb), Dst: JuniperN_d0:83:c2 (3c:8c:93:d0:83:c2)
Internet Protocol Version 4, Src: 10.16.22.173, Dst: 10.25.10.50
Transmission Control Protocol, Src Port: 5000, Dst Port: 60815, Seq: 68, Ack: 1, Len: 43
Data (43 bytes)
    Data: 636c69656e74730d0a0d0a69702031302e32352e31302e35300d0a69702031302e32362e…
    [Length: 43]

0000  3c 8c 93 d0 83 c2 18 c0  4d 3c 35 cb 08 00 45 00   <······ M<5···E·
0010  00 53 36 82 40 00 80 06  00 00 0a 10 16 ad 0a 19   ·S6·@··· ········
0020  0a 32 13 88 ed 8f d7 b3  53 37 b1 34 17 8c 50 18   ·2····· S7·4··P·
0030  04 02 35 4d 00 00 63 6c  69 65 6e 74 73 0d 0a 0d   ··5M··cl ients···
0040  0a 69 70 20 31 30 2e 32  35 2e 31 30 2e 35 30 0d   ·ip 10.2 5.10.50·
0050  0a 69 70 20 31 30 2e 32  36 2e 31 33 39 2e 32 32   ·ip 10.2 6.139.22
0060  36                                                6
```

4. Summary and future works

In this project, we implemented a simple online conference system, which supports real-time video transmission, real-time audio transmission, remote desktop sharing and control. In addition, we also designed a reasonable meeting management mechanism. In the process of code running, socket receiving and sending are prone to problems, resulting in the end of the program. After adding exception handling, the system can run more stably. Because the data is transmitted by TCP, there will be a certain lag in multi-party meetings, which has not been well solved. In a word, we learned a lot of useful knowledge about computer networks, which will be

beneficial to us in the long run. We will make further explorations in this area to gain a deeper understanding of computer networks.