

گزارش پروژه اول شبکه – پاییز 1403

محمدعلی قهاری 810100201

رضا عبدلی 810100251

سوالات تئوری:

WebRTC:

WebRTC یک فناوری برای ایجاد ارتباطات بلادرنگ (صدا، تصویر، و داده) بین کاربران بدون نیاز به نصب افزونه است. WebRTC به صورت پیش فرض از ارتباط P2P پشتیبانی می‌کند و برای کاربردهایی مانند تماس تصویری و صوتی مناسب است.

- مزایا: ارتباط مستقیم و بلادرنگ، بدون نیاز به افزونه.

- معایب: نیاز به راهکاری برای عبور از NAT و فایروال‌ها.

Coturn:

Coturn سروری است که پروتکل‌های STUN و TURN را اجرا می‌کند و برای حل مشکلات NAT و فایروال‌ها در ارتباطات WebRTC استفاده می‌شود. این سرور به دستگاه‌ها کمک می‌کند تا آدرس عمومی خود را تشخیص دهند یا داده‌ها را از طریق سرور واسطه ارسال کنند.

- مزایا: قابلیت اطمینان بیشتر در ارتباطات WebRTC حتی در شبکه‌های محدود.

- معایب: نیاز به منابع سرور، تأخیر بیشتر هنگام استفاده از TURN.

STUN:

پروتکل STUN به دستگاه‌ها کمک می‌کند تا آدرس عمومی خود را شناسایی کرده و برای ارتباط مستقیم P2P استفاده کنند.

- مزایا: کاهش تأخیر و منابع سرور به دلیل ارتباط مستقیم.

- معایب: در شبکه‌های پیچیده کارایی ندارد.

TURN:

پروتکل TURN زمانی استفاده می‌شود که ارتباط مستقیم ممکن نباشد. در این حالت، داده‌ها از طریق سرور TURN منتقل می‌شوند.

- مزایا: ارتباط پایدار حتی در شبکه‌های محدود.

- معایب: افزایش تأخیر و هزینه به دلیل عبور داده‌ها از سرور واسطه.

Signaling Server:

برای هماهنگی اولیه بین کاربران در WebRTC استفاده می‌شود. این سرور اطلاعاتی مانند IP، پورت و کلیدهای امنیتی را بین کاربران رد و بدل می‌کند تا ارتباط مستقیم برقرار شود. پس از برقراری ارتباط، دیگر نیازی به Signaling Server نیست.

- مزایا: ارتباط سریع‌تر: به کاربران کمک می‌کند تا سریع‌تر اطلاعات اولیه را رد و بدل کنند و ارتباط برقرار کنند

- معایب: نیاز به امنیت بالا: چون اطلاعات حساس اولیه از طریق آن رد و بدل می‌شود، باید از پروتکل‌های امنیتی قوی برای جلوگیری از دسترسی‌های غیرمجاز استفاده شود

بررسی webrtc

پروتکلی است که در آن دو کلاینت بدون نیاز به یک سرور برای ارتباط با آن، مستقیم با هم ارتباط می‌گیرند و پیام و اطلاعات خود را جابجا می‌کنند. در این روش، دو کلاینت ابتدا بوسیله یک signaling server با هم ارتباط می‌گیرند و SDP, candidate را با هم به اشتراک می‌گذارند. این ویژگی‌ها که در ادامه توضیح داده می‌شود، باعث می‌شود دو کلاینت با همدیگر مستقیم ارتباط بگیرند و سرور از ارتباط حذف شود. اینکار باعث می‌شود تاخیر به حداقل برسد.

ویژگی ice

یکی از مزیت‌های این پروتکل، ICE یا Interactive Connectivity Establishment است. ICE پروتکلی است که مشکلات NAT و فایروال را برای ارتباط peer-to-peer از میان برمی‌دارد. این روش بدین صورت عمل می‌کند که طرفین ارتباط با اشتراک گذاری مسیرهایی که می‌توانند با هم ارتباط برقرار کنند (که به هر کدام یک candidate گفته می‌شود) بهترین مسیر برای برقراری ارتباط را تشخیص می‌دهند و با حذف موانع ذکر شده در بالا با هم ارتباط را برقرار می‌کنند.

ویژگی sdp

پروتکل Session Description Protocol پروتکلی است که به وسیله آن امکان ارسال فایل‌های چند رسانه‌ای در بستر WebRTC امکان پذیر می‌شود. در واقع این پروتکل باعث ایجاد زبان مشترک بین دو دستگاه می‌شود و دو دستگاه با به اشتراک گذاری sdp هایشان با هم از نوع فایل‌ی که قرار است

ارسال شود، پروتکلی که تحت آن پیام ها ارسال می شود ، codec و سایر تنظیمات لازم برای برقراری ارتباط آگاه می شوند.

توضیح کد:

:WebRtc

```
Q_SIGNALS:
    void incomingPacket(const QByteArray &data, qint64 len);

    void incomingCall(const QString &username);

    void callRejected(const QString &peerID);

    void offerIsReady(const QString &peerID, const QString &description);

    void answerIsReady(const QString &peerID, const QString &description);

    void acceptedReceived();

    void rejectReceived();

    void endReceived();

    void isOffererChanged();

    void ssrcChanged();

    void payloadTypeChanged();

    void bitRateChanged();
```

این سیگنال های WebRtc هست که ما اونایی که استفاده نشدن رو حذف کردیم مثل زمانی که یک sdp ساخته میشه که کامل نیست(هنوز gathering state تغییر نکرده، چون ما فقط اون SDP آخر برامون مهمه که ارسالش کنیم)

بخشی از این سیگنال ها به کلاس سیگنال منیجر وصل میشه که کار های ارتباط با سیگنالینگ سرور رو هندل میکنه و اکثرشون به یوآی برنامه میرن و اولیشون هم برای صدا هست که از طریق اسپیکر پخش بشه.

کلیت کار اینجوریه که ابتدا با یک اسم داخل سیگنالینگ سرور رجیستر میشیم(اسم رو خودمون انتخاب میکنیم) بعد میتونیم اسم طرف مقابل رو وارد کنیم تا با اون ارتباط بگیریم و وقتی این کارو میکنیم peer طرف رو با اون اسمی که وارد کردیم میسازیم بعد که افر آماده شد از طریق سرور به طرف مقابل میدیم.

بعد طرف که درصافت کرد بهش پیام میدیم که قبول میکنه یا رد، اگه رد کرد که باز از طریق سرور پیام میدیم و همه چیو میبندیم ولی اگه قبول کرد اون طرف میاد peer این یکی رو میسازه و sdp دریافتی رو بهش الصاق میکنه تا answerer خودش ساخته بشه که بعد از ساختش اونو از طریق سرور ارسال میکنیم بعد در نهایت دو طرف به استیت 2 یعنی متصل میرن که اون موقع میایم میکروفون و اسپیکر

رو فعال میکنیم تا به محض آماده شدن پکت اونو ارسال و دریافت کنیم(بدون سرور).

```
void WebRTC::startCall(const QString &targetName)
{
    setIsOfferer(true);
    addPeer(targetName);
    addAudioTrack(targetName, "recv audio");
    generateOfferSDP(targetName);
}

void WebRTC::registerName(const QString &username)
{
    init(username);
    addPeer(username);
    m_signaller->connectToSignalingServer(username.toStdString());
}

void WebRTC::endCall(const QString &name)
{
    m_peerConnections[name]->close();
}

void WebRTC::acceptCall(const QString &name)
{
    if (!m_peerSdps.contains(name))
        return;
    rtc::Description desc(m_peerSdps[name].toStdString(), rtc::Description::Type::Offer);
    acceptPeer(desc, name);
}

void WebRTC::offerReceived(const QString &peerID, const QString &sdp)
{
    m_peerSdps.insert(peerID, sdp);
    setIsOfferer(false);
    Q_EMIT incommigCall(peerID);
}

void WebRTC::answerReceived(const QString &peerID, const QString &sdp)
{
    m_peerSdps.insert(peerID, sdp);
    rtc::Description desc(sdp.toStdString(), rtc::Description::Type::Answer);
    acceptPeer(desc, peerID);
    Q_EMIT acceptedReceived();
}
```

این وسط هم پکت ها از طریق track ها که قبل از جنریت کردن sdp ها ساخته میشن ارسال میشن
این طوری که افرر میاد ترک میسازه با مود sendrecv، بعد طرف مقابل هم میاد بعد از دریافت sdp
ترک میسازه

:AudioInput

کلاس AudioInput مسئول دریافت ورودی صوتی از میکروفون، پردازش و ارسال داده‌های صوتی از طریق کلاس WebRTC به کاربر مقصد(که در اینجا منظور AudioOutput می باشد) است.

```
AudioInput::AudioInput(WebRTC *webRtc, QObject *parent)
: QIODevice(parent)
, audioSource(nullptr)
, opusEncoder(nullptr)
{
    connect(this, &AudioInput::dataReady, webRtc, &WebRTC::dataReady, Qt::AutoConnection);
    QAudioFormat format;
    format.setSampleRate(44100);
    format.setChannelConfig(QAudioFormat::ChannelConfigMono);
    format.setSampleFormat(QAudioFormat::Int16);
    audioSource = new QAudioSource(format, this);

    int error;
    opusEncoder = opus_encoder_create(format.sampleRate(),
                                     format.channelCount(),
                                     OPUS_APPLICATION_VOIP,
                                     &error);

    // opusEncoder = opus_encoder_create(48000, 1, OPUS_APPLICATION_VOIP, &error);

    if (error != OPUS_OK)
        qWarning("Failed to create Opus encoder: %s", opus_strerror(error));
}
```

در این کد، یک سیگنال dataReady داریم که به اسلات dataReady در کلاس WebRTC متصل شده است. این به این معناست که هر زمان داده‌ی صوتی آماده ارسال شود (مثلاً در زمان فراخوانی تابع (writeData)، AudioInput سیگنالی به WebRTC می‌فرستد و به آن اطلاع می‌دهد که داده جدیدی آماده پردازش یا ارسال است.

در این متود فرمت صوتی، شامل Sample Rate، Channel Config و سپس منبع صوتی بر اساس فرمت تعریف شده را تنظیم میکنیم
در اینجا یک انکدر Opus برای کدگذاری نیز ایجاد کرده ایم اما به دلیل کُرش برنامه نتوانستیم از آن(در تابع writeData که در ادامه توضیح میدهیم) استفاده کنیم.

```

bool AudioInput::open(OpenMode mode)
{
    QAudioDevice inputDevice = QMediaDevices::defaultAudioInput();
    QAudioFormat format;
    format.setSampleRate(48000);
    format.setChannelConfig(QAudioFormat::ChannelConfigMono);
    format.setSampleFormat(QAudioFormat::Int16);

    if (!inputDevice.isFormatSupported(format)) {
        qWarning() << " selected format not supported"
                    << "ideal format:"
                    << inputDevice.preferredFormat();
        format = inputDevice.preferredFormat();
    }

    audioSource = new QAudioSource(format, this);

    if (!QIODevice::open(mode)) {
        qWarning() << "error opening AudioInput in WriteOnly";
        return false;
    }

    audioSource->start(this);
    return true;
}

void AudioInput::close()
{
    audioSource->stop();
    QIODevice::close();
}

```

در متود `open()` این کلاس ابتدا دستگاه ورودی صوتی پیش فرض (میکروفون) را تنظیم میکنیم. `(QMediaDevices::defaultAudioInput())` سپس با استفاده از `isFormatSupported` از سازگاری فرمت با میکروفون اطمینان حاصل میکنیم. اگر فرمت پشتیبانی نشود، از فرمت پیشنهادی سیستم استفاده میکنیم. سپس شروع به ضبط صوت با استفاده از `audioSource` و ارسال مستقیم داده‌ها به `writeData` میکنیم. در متود `close()` هم این تابع `audioSource` را متوقف کرده و `QIODevice` را می‌بندیم.

```

qint64 AudioInput::writeData(const char *data, qint64 len)
{
    // Emit the raw audio data as-is
    QByteArray byteArray(data, len);
    Q_EMIT dataReady(byteArray);

    return len;
}

```

در این نسخه از متود `writeData` که نوشتیم داده‌های صوتی بدون رمزگذاری ارسال میشوند. داده‌ها به

صورت خام به QByteArray تبدیل شده و از طریق سیگنال dataReady به WebRTC ارسال می‌شوند.

نسخه کامنت‌شده این تابع که شامل رمزگذاری Opus است پایین همین قطعه کد موجود است اما به دلیل کُرش توسط کالر (offerer). نتوانستیم از آن استفاده کنیم.

تابع readData() از کلاس AudioInput هم مورد نیاز نیست و پیاده‌سازی نمی‌شود، چون نیازی به خواندن داده از دستگاه خروجی نیست.

:AudioOutput

کلاس audiooutput برای دریافت و پخش صدای دریافتی از طرف مقابل (audioinput) طراحی شده است که از QAudioSink برای پخش صدا استفاده میکند. به دلیل مشابه در اینجا هم از opus استفاده نکردیم.

```
AudioOutput::AudioOutput(WebRTC *webRtc, QObject *parent)
: QObject(parent)
{
    connect(webRtc, &WebRTC::incomingPacket, this, &AudioOutput::receivedData, Qt::AutoConnection);

    QAudioDevice outputDevice = QMediaDevices::defaultAudioOutput();
    audioFormat.setSampleRate(44100);
    audioFormat.setChannelConfig(QAudioFormat::ChannelConfigMono);
    audioFormat.setSampleFormat(QAudioFormat::Int16);

    if (!outputDevice.isFormatSupported(audioFormat)) {
        qWarning() << "فرمت انتخابی پشتیبانی نمی‌شود. فرمت "
            "پیشنهادی:"
            << outputDevice.preferredFormat();
        audioFormat = outputDevice.preferredFormat();
    }

    int error;
    opus_decoder = opus_decoder_create(audioFormat.sampleRate(), audioFormat.channelCount(), &error);
    if (error != OPUS_OK) {
        qWarning() << "Failed to create Opus decoder:" << opus_strerror(error);
    }

    audioSink = new QAudioSink(audioFormat, this);
}
```

در constructor سیگنال incomingPacket از کلاس WebRTC را به اسلات receivedData در AudioOutput متصل می‌کند. این اتصال به این معناست که هر زمان داده صوتی دریافتی از شبکه آماده شد، این داده به AudioOutput منتقل و در receivedData پردازش می‌شود.

QMediaDevices::defaultAudioOutput() دستگاه صوتی خروجی پیش فرض را که بلندگو یا هدفون است، انتخاب می‌کند.

فرمت صوتی را نیز با نرخ نمونه برداری 441000 تنظیم کردیم که نشان دهنده تعداد نمونه هایی است که هر ثانیه از سیگنال صوتی گرفته میشود. فرمت داده ۱۶ بیتی نیز به این معنی است که هر نمونه از صدا با ۱۶ بیت داده ذخیره می‌شود. این حالت به کیفیت بالای صدا کمک میکند. در اینجا در صورتی که دستگاه خروجی از این فرمت پشتیبانی نکند، فرمت پیشنهادی دستگاه به جای آن استفاده می‌شود. در نهایت QAudioSink برای پخش صدا از دستگاه خروجی استفاده می‌شود و فرمت تنظیم شده را به عنوان پارامتر ورودی می‌گیرد.

```
//  
void AudioOutput::start()  
{  
    if (!device) {  
        device = audioSink->start(); // Start once and store the device for continuous playback  
        if (!device || !device->isOpen()) {  
            qWarning() << "Failed to start audio sink: device is not open.";  
        }  
    }  
}  
  
void AudioOutput::stop()  
{  
    if (audioSink->state() != QAudio::StoppedState) {  
        audioSink->stop();  
        device = nullptr; // Reset device handle on stop  
    }  
}
```

متود QAudioSink start() را راه اندازی کرده و دستگاه خروجی را برای پخش صدا آماده می‌کند. متد stop() هم پخش صدا را متوقف کرده و ابجکت دستگاه را به nullptr تنظیم می‌کند.


```

void AudioOutput::receivedData(const QByteArray &data, qint64 len)
{
    QMutexLocker locker(&mutex);

    if (device && device->isOpen())
        device->write(data.data(), len);
    else
        qWarning() << "Failed to write data to audio sink: device is not open.";
}

```

اسلات `receivedData()` داده‌ی دریافتی از WebRTC را دریافت و با استفاده از `device->write` به دستگاه صوتی ارسال می‌کند. در این قسمت باتوجه به اینکه `device->write(data.data(), len);` می‌تواند توسط Thread های مختلف همزمان فراخوانی شود، این بخش از `QMutexLocker` استفاده می‌کند تا مطمئن شود دسترسی به داده در این بخش از کد Synchronization ندارد و ممانعت از تداخل در دسترسی به داده‌ها را تضمین می‌کند.