iOS MCU Client Swift

Пример iOS-приложенияклиента для участника многоточечной конференции

Данный пример может использоваться для организации многоточечной видео конференции (MCU) на Web Call Server. Каждый участник такой конференции может публиковать WebRTC-поток и воспроизводить микшированный поток с аудио и видео других участников и собственным видео (без собственного аудио).

Для работы примера требуются следующие настройки в конфиге flashphoner.properties WCS-сервера

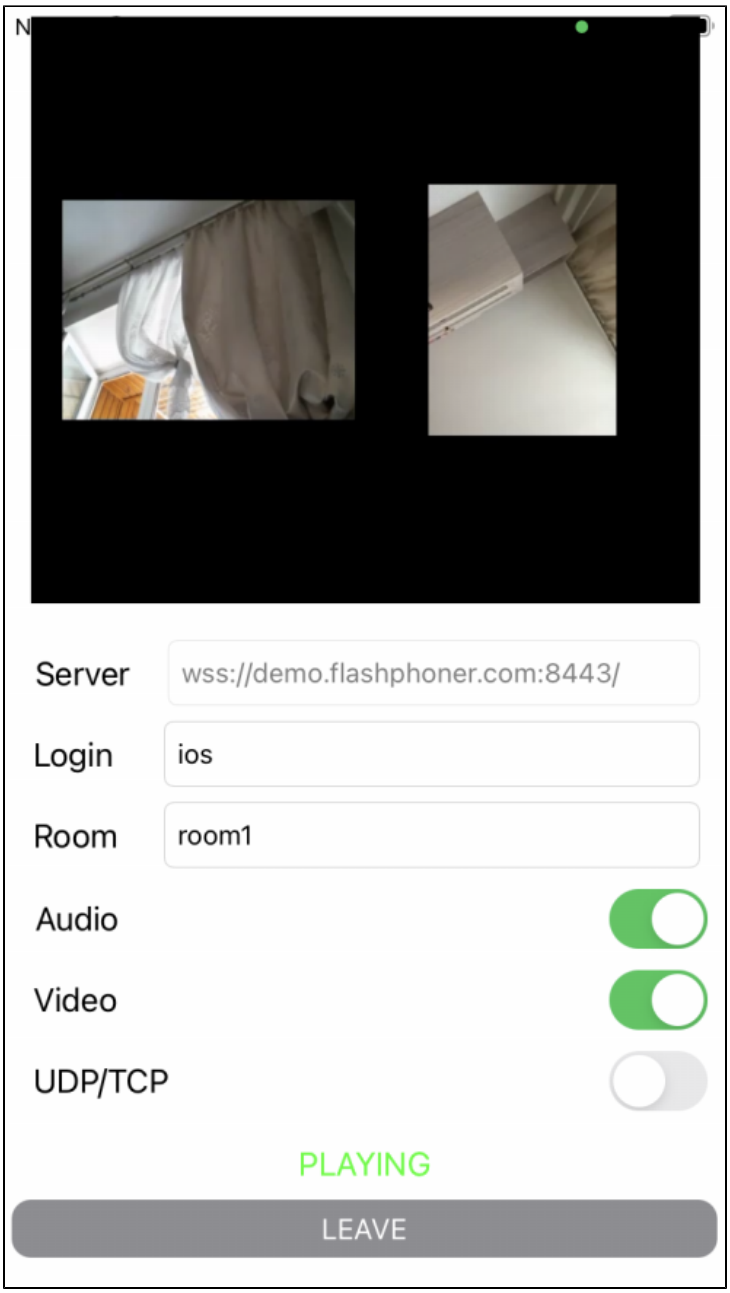
```
mixer_auto_start=true
mixer_mcu_audio=true
mixer_mcu_video=true
```

При подключении нового участника,использующего данный клиент, к конференции

- публикуется поток с видео участника и именем <participantName> + "#" + <roomName>

- этот поток добавляется к микшеру с именем <roomName> (если такой микшер еще не существует, то он создается)
- публикуется другой микшер с именем <roomName> + "-" + <participantName> + <roomName>, который содержит видео всех участников (включая данного) и аудио только от других участников, и начинается воспроизведение этого микшера

На скриншоте ниже участник конференции публикует поток и воспроизводит микшированный поток конференции:



Работа с кодом примера

Для разбора кода возьмем версию примера MCUClientSwift, которая доступна для скачивания на GitHub.

Класс для основного вида приложения: MCUViewController (файл имплементации MCUViewController.swift).

1. Импорт API. code

```
import FPWCSApi2Swift
```

2. Создание сессии и подключение к серверу.

WCSSession,WCSSession.connect code

В параметрах сессии указываются:

- URL WCS-сервера
- имя серверного приложения defaultApp

```
    @IBAction func joinPressed(_ sender: Any) {
        self.changeViewState(joinButton, false)
        if (joinButton.title(for: .normal) == "JOIN") {
            if (session == nil) {
                let options = FPWCSApi2SessionOptions()
                options.urlServer = serverField.text
                options.appKey = "defaultApp"
                do {
                    try session = WCSSession(options)
                } catch {
                    print(error)
                }
            }
            ...
            self.changeViewState(serverField, false)
            session?.connect()
        } else {
            leave()
        }

    }
```

3. Публикация видеопотока.

WCSSession.createStream, WCSStream.publish code

Методу createStream передаются параметры:

- имя публикуемого потока
- вид для локального отображения
- тип WebRTC транспорта
- констрейнты для публикации аудио и видео

```
    func publish() {
        ...
        let constraints = FPWCSApi2MediaConstraints()
        if (audioSwitch.isOn) {
            constraints.audio = FPWCSApi2AudioConstraints()
        }
        if (videoSwitch.isOn) {
            constraints.video = FPWCSApi2VideoConstraints()
        }
        let options = FPWCSApi2StreamOptions()
        options.name = loginField.text! + "#" + roomField.text!
        options.transport = transportSwitch.isOn ? kFPWCSTransport.fpwcsTransportTCP : kFPWCSTransport.
fpwcsTransportUDP
        options.constraints = constraints
        options.display = localDisplay.videoView
        do {
            publishStream = try session!.createStream(options)
        } catch {
            print(error);
        }
        ...
        do {
            try publishStream?.publish()
        } catch {
            print(error);
        }
    }
```

4. Воспроизведение видеопотока.

WCSSession.createStream, WCSStream.play code

Методу createStream передаются параметры:

- имя воспроизводимого потока
- вид для отображения потока
- тип WebRTC транспорта

```
    func play() {
        ...
        let options = FPWCSApi2StreamOptions()
        options.name = roomField.text! + "-" + loginField.text! + roomField.text!
        options.transport = transportSwitch.isOn ? kFPWCSTransport.fpwcsTransportTCP : kFPWCSTransport.
fpwcsTransportUDP
        options.display = remoteDisplay.videoView;
        do {
            playStream = try session!.createStream(options)
        } catch {
            print(error)
        }
        ...
        do {
            try playStream?.play()
        } catch {
            print(error);
        }
    }
```

5. Остановка воспроизведения видеопотока.

WCSStream.stop code

```
    func stopPlay() {
        do {
            try playStream?.stop();
        } catch {
            print(error);
        }
        playStream = nil;
    }
```

6. Остановка публикации видеопотока.

WCSStream.stop code

```
    func stopPublish() {
        do {
            try publishStream?.stop()
        } catch {
            print(error);
        }
        publishStream = nil;
    }
```

7. Закрытие соединения.

WCSSession.disconnect code

```
    func leave() {
        session?.disconnect()
        session = nil;
    }
```