Если вы видите что-то необычное, просто сообщите мне.

Building a WebRTC video and audio Broadcaster in Golang using ION-SFU, and media devices

Gabriel Tanne Gabriel Tanner

Building a WebRTC video and audio Broadcaster in Golang using ION-SFU, and media devices

Table of Contents

In this tutorial, you will build a video broadcasting application that reads the camera in Golang and sends it to the ION-SFU (Selective forwarding unit) which allows WebRTC sessions to scale more efficiently.

WebRTC, short for Web Real-Time Communication, is a communication protocol that enables real-time audio, video and data transmission on the web by utilizing peer to peer connections.

WebRTC also provides a Javascript API that is available by default in most browsers and helps developers implement the protocol in their applications. But there are also some implementations of the WebRTC protocol in other languages.

In this tutorial, you will build a video broadcasting application that reads the camera in Golang and sends it to the ION-SFU (Selective forwarding unit) which allows WebRTC sessions to scale more efficiently.

The application will also feature a small frontend that lets you watch the video you published by reading it from the ION-SFU server.

Prerequisites

Before you begin this guide, you'll need the following:

A valid Golang installation.

Camera connected to your computer that can be read using Video for Linux as a source for the video stream.

(Optional) If you want to connect with devices that are not on your network you will need to add a TURN server to your application. If you want to know more about TURN and how to set up your own check out this article.

Technology Stack

Now that you have an overview of what you are going to build let's take a closer look at the tools in use and how they work with each other.

Let's break the different components down:

Pion - Pure Golang implementation of the WebRTC protocol. Used to establish a peer connection to ION-SFU and send the video stream.

ION SFU - ION SFU (Selective Forwarding Unit) is a video routing service that allows Webrtc sessions to scale more efficiently.

Pion mediadevices - Golang implementation of the Mediadevices API which is used to read the camera as a Mediastream that can be sent using the peer connection.

One main benefit of this is that you can read the camera without the need to open a browser tab. Using a selective forwarding unit will also help a lot with performance and scaling the application for a large size of users.

This article assumes a basic knowledge of WebRTC. If you do not have any previous experience, I recommend reading the free book WebRTC for the curious.

Setting up ION-SFU

In this section, you will clone and configure the ION-SFU server so that you can use it with your application.

First, you will clone the repository so you have all the resources needed to start setting up your selective forwarding unit:

git clone --branch v1.10.6 https://github.com/pion/ion-sfu.git

This command will clone the ION-SFU repository from Github and create a folder with the name of ion-sfu in your directory. Now enter the directory using the following command:

cd ion-sfu

Next you can edit the configuration of the sfu by changing the config.toml file. The standard configurations are fine for testing and local use but I would recommend adding a STUN and TURN server if you try to access the server from a device in another network.

If you are not sure how to create a TURN server I would recommend reading this guide.

Once you are done with the configuration you can start the server using the following command:

go build ./cmd/signal/json-rpc/main.go && ./main -c config.toml

Alternatively you can also start the server using Docker if you prefer that over starting it using Golang.

docker run -p 7000:7000 -p 5000-5020:5000-5020/udp pionwebrtc/ion-sfu:v1.10.6-jsonrpc

You have now successfully set up your ION-SFU server and should see the following output in the console.

config config.toml load ok!

[2020-10-12 19:04:19.017] [INFO] [376][main.go][main] => --- Starting SFU Node ---

[2020-10-12 19:04:19.018] [INFO] [410][main.go][main] => Listening at http://[:7000]

Creating the project

Now that the setup and configuration of the ion-sfu server are done it is time to create the project

First, you will need to create a directory and enter it.

mkdir mediadevice-broadcast && cd mediadevice-broadcast

After that you can continue by creating all the files needed for the project using the following command:

mkdir public

touch main.go public/index.html public/index.js public/style.css

There are also two packages that need to be installed to follow this article.

sudo apt-get install -y v4l-utils

sudo apt-get install -y libvpx-dev

If you are not on Linux you might need to download different packages. Look at the media devices documentation for more information.

Establishing a WebRTC connection

Before any data can be exchanged using WebRTC, there must first be an established peer-to-peer connection between two WebRTC agents. Since the peer-to-peer connection often cannot be established directly there needs to be some signaling method.

Signaling to the ion-sfu will be handled over the Websockets protocol. For that, we will implement a simple Websockets boilerplate using the gorilla/websocket library that connects to the Websockets server and allows us to receive the incoming message and send our own.

```
package main

import (
```

```
"bytes"

"encoding/json"

"flag"

"fmt"

"io"

"log"

"net/url"

"github.com/google/uuid"

"github.com/gorilla/websocket"
```

)

var addr string

func main() {

```
flag.StringVar(&addr, "a", "localhost:7000", "address to use")

flag.Parse()

u := url.URL{Scheme: "ws", Host: addr, Path: "/ws"}

log.Printf("connecting to %s", u.String())

c, _, err := websocket.DefaultDialer.Dial(u.String(), nil)

if err != nil {

log.Fatal("dial:", err)

}
```

```
 defer c.Close()

 // Read incoming Websocket messages

 done := make(chan struct{})

 go readMessage(c, done)

 <-done
```

}

func readMessage(connection *websocket.Conn, done chan struct{}) {

```
 defer close(done)

 for {

 _, message, err := connection.ReadMessage()

 if err != nil || err == io.EOF {

 log.Fatal("Error reading: ", err)

 break

 }

 fmt.Printf("recv: %s", message)

 }
```

}

Now let's walk through the code for better understanding:

The flag is used to dynamically provide the URL of the Websockets server when starting the script and has a standard value of localhost:7000

The URL is used to create a Websockets client using the Dial method. Then we check if the connection resulted in an error and print a log if that is the case.

The readMessage function then reads the incoming messages by calling ReadMessage() on the Websocket connection and is run as a Go routine so it doesn't block the main thread and can run in the background.

The last line of the main() function makes sure that the script runs as long as the done variable is not closed.

The next step is creating a peer connection to the ion-sfu and handling the incoming WebRTC signaling events.

var peerConnection *webrtc.PeerConnection

func main() {

...

```
config := webrtc.Configuration{

□ICEServers: []webrtc.ICEServer{

□□{

□□□URLs: []string{"stun:stun.l.google.com:19302"},

□□},

□□/*{

□□□URLs:       []string{"turn:TURN_IP:3478?transport=tcp"},

□□□Username:   "username",

□□□Credential: "password",

□□},*/
```

```
	},
	SDPSemantics: webrtc.SDPSemanticsUnifiedPlanWithFallback,
	}
	// Create a new RTCPeerConnection
	mediaEngine := webrtc.MediaEngine{}
	vpxParams, err := vpx.NewVP8Params()
	if err != nil {
	panic(err)
	}
	vpxParams.BitRate = 500_000 // 500kbps
	codecSelector := mediadevices.NewCodecSelector(
	mediadevices.WithVideoEncoders(&vpxParams),
	)
	codecSelector.Populate(&mediaEngine)
	api := webrtc.NewAPI(webrtc.WithMediaEngine(mediaEngine))
	peerConnection, err = api.NewPeerConnection(config)
	if err != nil {
	panic(err)
	}
}
```

Here we first create a WebRTC config where we define our STUN and TURN server that will be used in the signaling process. After, that we create a MediaEngine that lets us define the codecs supported by the peer connection.

With all that configuration done we can create a new peer connection by calling the NewPeerConnection function on the WebRTC API we just created.

Before sending the offer to the ion-sfu server over Websockets we first need to add the video and audio stream. This is where the media device library comes into play to read the video from the camera.

```
fmt.Println(mediadevices.EnumerateDevices())

s, err := mediadevices.GetUserMedia(mediadevices.MediaStreamConstraints{

	Video: func(c *mediadevices.MediaTrackConstraints) {

		c.FrameFormat = prop.FrameFormat(frame.FormatYUY2)

		c.Width = prop.Int(640)

		c.Height = prop.Int(480)

	},

	Codec: codecSelector,

})

if err != nil {

	panic(err)

}

for _, track := range s.GetTracks() {

	track.OnEnded(func(err error) {
```

```
    fmt.Printf("Track (ID: %s) ended with error: %v\n",
        track.ID(), err)
  })
  _, err = peerConnection.AddTransceiverFromTrack(track,
    webrtc.RtpTransceiverInit{
      Direction: webrtc.RTPTransceiverDirectionSendonly,
    },
  )
  if err != nil {
    panic(err)
  }
}
```

Once an instance of the media devices library is created using the peer connection you can get the user media using the GetUserMedia function and passing the parameters.

One configuration change you might need to make is altering the FrameFormat to support your connected camera. You can check the frame format of your camera with the following command:

v4l2-ctl --all

All supported formats can also be found in the media devices Github repository.

The offer can now be created and saved into the local description of the peer connection.

```
// Creating WebRTC offer
offer, err := peerConnection.CreateOffer(nil)
```

```
// Set the remote SessionDescription

err = peerConnection.SetLocalDescription(offer)

if err != nil {

panic(err)

}
```

The next step is to send the offer over to the sfu using Websockets. The Websockets message is JSON and needs a specific structure to be recognized by the sfu.

Therefore we need to create a struct holding our offer and the required sid that specifies the room we want to join that we can then convert into JSON.

type SendOffer struct {

```
SID   string                     `json:sid`

Offer *webrtc.SessionDescription `json:offer`
```

}

Now we convert our offer object into JSON using the json.Marshal() function and then use the JSON offer object as a parameter in the request.

After converting the request to a byte array the message can finally be send over Websockets using the WriteMessage() function.

```
offerJSON, err := json.Marshal(&SendOffer{

Offer: peerConnection.LocalDescription(),

SID:   "test room",

})

params := (*json.RawMessage)(&offerJSON)
```

```
connectionUUID := uuid.New()

connectionID = uint64(connectionUUID.ID())

offerMessage := &jsonrpc2.Request{

	Method: "join",

	Params: params,

	ID: jsonrpc2.ID{

		IsString: false,

		Str:      "",

		Num:      connectionID,

	},

}

reqBodyBytes := new(bytes.Buffer)

json.NewEncoder(reqBodyBytes).Encode(offerMessage)

messageBytes := reqBodyBytes.Bytes()

c.WriteMessage(websocket.TextMessage, messageBytes)
```

Now that the offer is sent we need to correctly respond to the WebRTC events and the response from the Websockets server.

The OnICECandidate event is called whenever a new ICE candidate is found. The method is then used to negotiate a connection with the remote peer by sending a trickle request to the sfu.

```
// Handling OnICECandidate event

peerConnection.OnICECandidate(func(candidate *webrtc.ICECandidate) {
```

```
    if candidate != nil {
        candidateJSON, err := json.Marshal(&Candidate{
            Candidate: candidate,
            Target: 0,
        })
        params := (*json.RawMessage)(&candidateJSON)
        if err != nil {
            log.Fatal(err)
        }
        message := &jsonrpc2.Request{
            Method: "trickle",
            Params: params,
        }
        reqBodyBytes := new(bytes.Buffer)
        json.NewEncoder(reqBodyBytes).Encode(message)
        messageBytes := reqBodyBytes.Bytes()
        c.WriteMessage(websocket.TextMessage, messageBytes)
    }
})
peerConnection.OnICEConnectionStateChange(func(connectionState webrtc.ICEConnectionState) {
```

```
□fmt.Printf("Connection State has changed to %s \n", connectionState.String())

})
```

The readMessage function created earlier is used to receive and react to the incoming Websockets messages send by the sfu.

For that we first need to create the structs that contain the received messages so we can use the data. Then we will determine which event the message is for and handle them accordingly.

// SendAnswer object to send to the sfu over Websockets

type SendAnswer struct {

```
SID    string                    `json:sid`

Answer *webrtc.SessionDescription `json:answer`
```

}

type ResponseCandidate struct {

```
Target    int                   `json:"target"`

Candidate *webrtc.ICECandidateInit `json:candidate`
```

}

// TrickleResponse received from the sfu server

type TrickleResponse struct {

```
Params ResponseCandidate□□□□`json:params`

Method string                  `json:method`
```

}

// Response received from the sfu over Websockets

type Response struct {

```
Params *webrtc.SessionDescription `json:params`

Result *webrtc.SessionDescription `json:result`

Method string                     `json:method`

Id     uint64                     `json:id`
```

}

func readMessage(connection *websocket.Conn, done chan struct{}) {

```
defer close(done)

for {

□_, message, err := connection.ReadMessage()

□if err != nil || err == io.EOF {

□□log.Fatal("Error reading: ", err)

□□break

□}

□fmt.Printf("recv: %s", message)

□var response Response

□json.Unmarshal(message, &response)

□if response.Id == connectionID {

□□result := *response.Result

□□remoteDescription = response.Result
```

```
□□if err := peerConnection.SetRemoteDescription(result); err != nil {
□□□log.Fatal(err)
□□}
□} else if response.Id != 0 && response.Method == "offer" {
□□peerConnection.SetRemoteDescription(*response.Params)
□□answer, err := peerConnection.CreateAnswer(nil)
□□if err != nil {
□□□log.Fatal(err)
□□}
□□peerConnection.SetLocalDescription(answer)
□□connectionUUID := uuid.New()
□□connectionID = uint64(connectionUUID.ID())
□□offerJSON, err := json.Marshal(&SendAnswer{
□□□Answer: peerConnection.LocalDescription(),
□□□SID:    "test room",
□□})
□□params := (*json.RawMessage)(&offerJSON)
□□answerMessage := &jsonrpc2.Request{
□□□Method: "answer",
□□□Params: params,
```

```
□□□ID: jsonrpc2.ID{
□□□□IsString: false,
□□□□Str:      "",
□□□□Num:      connectionID,
□□□},
□□}
□□reqBodyBytes := new(bytes.Buffer)
□□json.NewEncoder(reqBodyBytes).Encode(answerMessage)
□□messageBytes := reqBodyBytes.Bytes()
□□connection.WriteMessage(websocket.TextMessage, messageBytes)
□} else if response.Method == "trickle" {
□□var trickleResponse TrickleResponse
□□if err := json.Unmarshal(message, &trickleResponse); err != nil {
□□□log.Fatal(err)
□□}
□□err := peerConnection.AddICECandidate(*trickleResponse.Params.Candidate)
□□if err != nil {
□□□log.Fatal(err)
□□}
□}
```

```
}
```

}

As you can see we are handling two different events:

Offer - The sfu sends an offer and we react by saving the send offer into the remote description of our peer connection and sending back an answer with the local description so we can connect to the remote peer.

Trickle - The sfu sends a new ICE candidate and we add it to the peer connection

All this configuration will result in the following file:

package main

import (

```
"bytes"

"encoding/json"

"flag"

"fmt"

"io"

"log"

"net/url"

"github.com/google/uuid"

"github.com/gorilla/websocket"

"github.com/pion/mediadevices"

"github.com/pion/mediadevices/pkg/codec/vpx"
```

```
"github.com/pion/mediadevices/pkg/frame"

"github.com/pion/mediadevices/pkg/prop"

"github.com/pion/webrtc/v3"

"github.com/sourcegraph/jsonrpc2"

// Note: If you don't have a camera or microphone or your adapters are not supported,

//       you can always swap your adapters with our dummy adapters below.

// _ "github.com/pion/mediadevices/pkg/driver/videotest"

// _ "github.com/pion/mediadevices/pkg/driver/audiotest"

_ "github.com/pion/mediadevices/pkg/driver/camera"     // This is required to register camera
adapter

_ "github.com/pion/mediadevices/pkg/driver/microphone" // This is required to register
microphone adapter
```

)

type Candidate struct {

```
Target    int                  `json:"target"`

Candidate *webrtc.ICECandidate `json:candidate`
```

}

type ResponseCandidate struct {

```
Target    int                  `json:"target"`

Candidate *webrtc.ICECandidateInit `json:candidate`
```

}

// SendOffer object to send to the sfu over Websockets

type SendOffer struct {

```
SID   string                     `json:sid`

Offer *webrtc.SessionDescription `json:offer`
```

}

// SendAnswer object to send to the sfu over Websockets

type SendAnswer struct {

```
SID    string                     `json:sid`

Answer *webrtc.SessionDescription `json:answer`
```

}

// TrickleResponse received from the sfu server

type TrickleResponse struct {

```
Params ResponseCandidate□□□□`json:params`

Method string                   `json:method`
```

}

// Response received from the sfu over Websockets

type Response struct {

```
Params *webrtc.SessionDescription `json:params`

Result *webrtc.SessionDescription `json:result`

Method string                     `json:method`
```

```
Id     uint64                    `json:id`
```

}

var peerConnection *webrtc.PeerConnection

var connectionID uint64

var remoteDescription *webrtc.SessionDescription

var addr string

func main() {

```
flag.StringVar(&addr, "a", "localhost:7000", "address to use")

flag.Parse()

u := url.URL{Scheme: "ws", Host: addr, Path: "/ws"}

log.Printf("connecting to %s", u.String())

c, _, err := websocket.DefaultDialer.Dial(u.String(), nil)

if err != nil {

log.Fatal("dial:", err)

}

defer c.Close()

config := webrtc.Configuration{

ICEServers: []webrtc.ICEServer{

{

URLs: []string{"stun:stun.l.google.com:19302"},
```

```
□□},
□□/*{
□□□URLs:       []string{"turn:TURN_IP:3478"},
□□□Username:   "username",
□□□Credential: "password",
□□},*/
□},
□SDPSemantics: webrtc.SDPSemanticsUnifiedPlanWithFallback,
}
// Create a new RTCPeerConnection
mediaEngine := webrtc.MediaEngine{}
vpxParams, err := vpx.NewVP8Params()
if err != nil {
□panic(err)
}
vpxParams.BitRate = 500_000 // 500kbps
codecSelector := mediadevices.NewCodecSelector(
□mediadevices.WithVideoEncoders(&vpxParams),
)
codecSelector.Populate(&mediaEngine)
```

```
api := webrtc.NewAPI(webrtc.WithMediaEngine(&mediaEngine))

peerConnection, err = api.NewPeerConnection(config)

if err != nil {

panic(err)

}

// Read incoming Websocket messages

done := make(chan struct{})

go readMessage(c, done)

fmt.Println(mediadevices.EnumerateDevices())

s, err := mediadevices.GetUserMedia(mediadevices.MediaStreamConstraints{

Video: func(c *mediadevices.MediaTrackConstraints) {

c.FrameFormat = prop.FrameFormat(frame.FormatYUY2)

c.Width = prop.Int(640)

c.Height = prop.Int(480)

},

Codec: codecSelector,

})

if err != nil {

panic(err)

}
```

```
for _, track := range s.GetTracks() {

	track.OnEnded(func(err error) {

		fmt.Printf("Track (ID: %s) ended with error: %v\n",

			track.ID(), err)

	})

	_, err = peerConnection.AddTransceiverFromTrack(track,

		webrtc.RtpTransceiverInit{

			Direction: webrtc.RTPTransceiverDirectionSendonly,

		},

	)

	if err != nil {

		panic(err)

	}

}

// Creating WebRTC offer

offer, err := peerConnection.CreateOffer(nil)

// Set the remote SessionDescription

err = peerConnection.SetLocalDescription(offer)

if err != nil {

	panic(err)
```

```
}

// Handling OnICECandidate event

peerConnection.OnICECandidate(func(candidate *webrtc.ICECandidate) {

	if candidate != nil {

		candidateJSON, err := json.Marshal(&Candidate{

			Candidate: candidate,

			Target: 0,

		})

		params := (*json.RawMessage)(&candidateJSON)

		if err != nil {

			log.Fatal(err)

		}

		message := &jsonrpc2.Request{

			Method: "trickle",

			Params: params,

		}

		reqBodyBytes := new(bytes.Buffer)

		json.NewEncoder(reqBodyBytes).Encode(message)

		messageBytes := reqBodyBytes.Bytes()

		c.WriteMessage(websocket.TextMessage, messageBytes)
```

```
	}
	})
	peerConnection.OnICEConnectionStateChange(func(connectionState webrtc.ICEConnectionState) {
	fmt.Printf("Connection State has changed to %s \n", connectionState.String())
	})
	offerJSON, err := json.Marshal(&SendOffer{
	Offer: peerConnection.LocalDescription(),
	SID:   "test room",
	})
	params := (*json.RawMessage)(&offerJSON)
	connectionUUID := uuid.New()
	connectionID = uint64(connectionUUID.ID())
	offerMessage := &jsonrpc2.Request{
	Method: "join",
	Params: params,
	ID: jsonrpc2.ID{
		IsString: false,
		Str:      "",
		Num:      connectionID,
	},
```

```
}

reqBodyBytes := new(bytes.Buffer)

json.NewEncoder(reqBodyBytes).Encode(offerMessage)

messageBytes := reqBodyBytes.Bytes()

c.WriteMessage(websocket.TextMessage, messageBytes)

<-done
```

}

func readMessage(connection *websocket.Conn, done chan struct{}) {

```
defer close(done)

for {

□_, message, err := connection.ReadMessage()

□if err != nil || err == io.EOF {

□□log.Fatal("Error reading: ", err)

□□break

□}

□fmt.Printf("recv: %s", message)

□var response Response

□json.Unmarshal(message, &response)

□if response.Id == connectionID {

□□result := *response.Result
```

```
□□remoteDescription = response.Result
□□if err := peerConnection.SetRemoteDescription(result); err != nil {
□□□log.Fatal(err)
□□}
□} else if response.Id != 0 && response.Method == "offer" {
□□peerConnection.SetRemoteDescription(*response.Params)
□□answer, err := peerConnection.CreateAnswer(nil)
□□if err != nil {
□□□log.Fatal(err)
□□}
□□peerConnection.SetLocalDescription(answer)
□□connectionUUID := uuid.New()
□□connectionID = uint64(connectionUUID.ID())
□□offerJSON, err := json.Marshal(&SendAnswer{
□□□Answer: peerConnection.LocalDescription(),
□□□SID:    "test room",
□□})
□□params := (*json.RawMessage)(&offerJSON)
□□answerMessage := &jsonrpc2.Request{
□□□Method: "answer",
```

```
□□□Params: params,
□□□ID: jsonrpc2.ID{
□□□□IsString: false,
□□□□Str:      "",
□□□□Num:      connectionID,
□□□},
□□}
□□reqBodyBytes := new(bytes.Buffer)
□□json.NewEncoder(reqBodyBytes).Encode(answerMessage)
□□messageBytes := reqBodyBytes.Bytes()
□□connection.WriteMessage(websocket.TextMessage, messageBytes)
□} else if response.Method == "trickle" {
□□var trickleResponse TrickleResponse
□□if err := json.Unmarshal(message, &trickleResponse); err != nil {
□□□log.Fatal(err)
□□}
□□err := peerConnection.AddICECandidate(*trickleResponse.Params.Candidate)
□□if err != nil {
□□□log.Fatal(err)
□□}
```

```
□}

}
```

}

Note: You might need to enable go modules so that the dependencies are downloaded automatically when starting the script.

The finished script can now be started using the following command:

You might need to add sudo to access your camera

go run main.go

You should see the following output:

recv: {"method":"trickle","params":{"candidate":"candidate:3681230645 1 udp 2130706431 10.0.0.35 49473 typ host","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

recv: {"method":"trickle","params":{"candidate":"candidate:233762139 1 udp 2130706431 172.17.0.1 57218 typ host","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

Connection State has changed to checking

recv: {"method":"trickle","params":{"candidate":"candidate:2890797847 1 udp 2130706431 172.22.0.1 41179 typ host","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

recv: {"method":"trickle","params":{"candidate":"candidate:3528925834 1 udp 2130706431 172.18.0.1 58906 typ host","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

recv: {"method":"trickle","params":{"candidate":"candidate:3197649470 1 udp 1694498815 212.197.155.248 36942 typ srflx raddr 0.0.0.0 rport 36942","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

recv: {"method":"trickle","params":{"candidate":"candidate:2563076625 1 udp 16777215 104.248.140.156 11643 typ relay raddr 0.0.0.0 rport 42598","sdpMid":"","sdpMLineIndex":0,"usernameFragment":null},"jsonrpc":"2.0"}

Connection State has changed to connected

Client-side

Now that we are successfully sending the video from the camera to the sfu it is time to create a frontend to receive it.

The HTML file is very basic and will only contain a video object and a button to subscribe to the stream. It will also print the current WebRTC logs into a div.

```
<meta charset="utf-8"/>

<meta

        name="viewport"

        content="width=device-width, initial-scale=1, shrink-to-fit=no"

/>

<style>

    #remotes video {

        width: 320px;

    }

</style>

<title>WebRTC test frontend</title>
```

```
<div id="remotes">

      <span

              style="position: absolute; margin-left: 5px; margin-top: 5px"

              class="badge badge-primary"

      >Remotes</span

      >

</div>
```

The javascript file will then connect to the sfu similar to the Golang script above. The only difference is that instead of reading the camera and sending the video to the sfu it will receive the video instead.

I will not go into much detail since all the functionality is already covered above.

const remotesDiv = document.getElementById("remotes");

const config = {

```
codec: 'vp8',

iceServers: [

    {

        "urls": "stun:stun.l.google.com:19302",

    },

    /*{

        "urls": "turn:TURN_IP:3468",

        "username": "username",
```

```
        "credential": "password"
    },*/
]
```

};

const signalLocal = new Signal.IonSFUJSONRPCSignal(

```
"ws://127.0.0.1:7000/ws"
```

);

const clientLocal = new IonSDK.Client(signalLocal, config);

signalLocal.onopen = () => clientLocal.join("test room");

clientLocal.ontrack = (track, stream) => {

```
console.log("got track", track.id, "for stream", stream.id);
if (track.kind === "video") {
    track.onunmute = () => {
        const remoteVideo = document.createElement("video");
        remoteVideo.srcObject = stream;
        remoteVideo.autoplay = true;
        remoteVideo.muted = true;
        remotesDiv.appendChild(remoteVideo);
        track.onremovetrack = () => remotesDiv.removeChild(remoteVideo);
    };
```

```
  }
```

};

The only thing you have to keep in mind here is that a peer connection cannot be sent without offering or requesting some kind of stream. That is why add two receivers (one for audio and one for video) before sending the offer.

You can now start the frontend by opening the HTML file in your browser. Alternatively, you can open the HTML file using an Express server by creating a new file in the project's root directory.

touch server.js

The express dependency needs to be installed before adding the code.

npm init -y

npm install express --save

Then you can run your frontend as a static site using the following code.

const express = require("express");

const app = express();

const port = 3000;

const http = require("http");

const server = http.createServer(app);

app.use(express.static(__dirname + "/public"));

server.listen(port, () => console.log(`Server is running on port ${port}`));

Start the application using the following command.

node server.js

You should now be able to access the frontend on localhost:3000 of your local machine.

Ion-SDK-Go

As mentioned above, the same video streaming functionality can also be implemented using a helper library created by the ion team, which abstracts the WebRTC signaling and therefore makes the implementation shorter and more concise. Still, knowing how to implement the signaling yourself is very important and leaves you with more customization for more complex projects.

package main

import (

```
"flag"

"fmt"

ilog "github.com/pion/ion-log"

sdk "github.com/pion/ion-sdk-go"

"github.com/pion/mediadevices"

"github.com/pion/mediadevices/pkg/codec/vpx"

"github.com/pion/mediadevices/pkg/frame"

"github.com/pion/mediadevices/pkg/prop"

"github.com/pion/webrtc/v3"

// Note: If you don't have a camera or microphone or your adapters are not supported,

//       you can always swap your adapters with our dummy adapters below.

// _ "github.com/pion/mediadevices/pkg/driver/videotest"

// _ "github.com/pion/mediadevices/pkg/driver/audiotest"
```

```
_ "github.com/pion/mediadevices/pkg/driver/camera"     // This is required to register camera adapter

_ "github.com/pion/mediadevices/pkg/driver/microphone" // This is required to register microphone adapter
```

)

var (

```
log = ilog.NewLoggerWithFields(ilog.DebugLevel, "", nil)
```

)

func main() {

```
// parse flag

var session, addr string

flag.StringVar(&addr, "addr", "localhost:50051", "Ion-sfu grpc addr")

flag.StringVar(&session, "session", "test room", "join session name")

flag.Parse()

// add stun servers

webrtcCfg := webrtc.Configuration{

□ICEServers: []webrtc.ICEServer{

□□webrtc.ICEServer{

□□□URLs: []string{"stun:stun.stunprotocol.org:3478", "stun:stun.l.google.com:19302"},

□□},

□},
```

```
}

config := sdk.Config{

Log: log.Config{

Level: "debug",

},

WebRTC: sdk.WebRTCTransportConfig{

Configuration: webrtcCfg,

},

}

// new sdk engine

e := sdk.NewEngine(config)

// get a client from engine

c, err := sdk.NewClient(e, addr, "client id")

c.GetPubTransport().GetPeerConnection().OnICEConnectionStateChange(func(state
webrtc.ICEConnectionState) {

log.Infof("Connection state changed: %s", state)

})

if err != nil {

log.Errorf("client err=%v", err)

panic(err)

}
```

```
e.AddClient(c)

// client join a session

err = c.Join(session, nil)

if err != nil {

	log.Errorf("join err=%v", err)

	panic(err)

}

vpxParams, err := vpx.NewVP8Params()

if err != nil {

	panic(err)

}

vpxParams.BitRate = 500_000 // 500kbps

codecSelector := mediadevices.NewCodecSelector(

	mediadevices.WithVideoEncoders(&vpxParams),

)

fmt.Println(mediadevices.EnumerateDevices())

s, err := mediadevices.GetUserMedia(mediadevices.MediaStreamConstraints{

	Video: func(c *mediadevices.MediaTrackConstraints) {

		c.FrameFormat = prop.FrameFormat(frame.FormatYUY2)

		c.Width = prop.Int(640)
```

```
		c.Height = prop.Int(480)
	},
	Codec: codecSelector,
})
if err != nil {
	panic(err)
}
for _, track := range s.GetTracks() {
	track.OnEnded(func(err error) {
		fmt.Printf("Track (ID: %s) ended with error: %v\n",
			track.ID(), err)
	})
	_, err = c.Publish(track)
	if err != nil {
		panic(err)
	} else {
		break // only publish first track, thanks
	}
}
select {}
```

```
}
```

As you can see, the library handles the signaling and therefore abstracts a lot of the code we wrote above. One difference between the library and the code we implemented above is that the library uses GRPC for signaling, whereas we use JSONRPC. You will therefore have to start ion-sfu in AllRPC mode instead of JSONRPC. This can be done by either starting the AllRPC version using Golang or by using the AllRPC image tag (e.g. latest-allrpc) when using Docker.

You can now start the application using the go run command:

```
go run ./main.go
```

You should now also be able to see the video of your camera on localhost:3000 of your local machine.

Conclusion

In this article, you learned what a sfu is and how you can utilize the ion-sfu to build a video broadcasting application. You also learned how to use the Golang media device library to read your camera without opening a browser window.

If you are interested in learning more, consider subscribing to my email list to never miss another article. Also, feel free to leave some feedback or check out my other articles.

Revision #1
Created 2023-02-07 21:57:28 UTC by gasick
Updated 2023-04-16 19:36:18 UTC by gasick