**Tutorial WebRTC**

**Versão PDF Guia rápido Recursos Pesquisa de emprego Discussão**

**Com Web Real-Time Communication (WebRTC), modernas aplicações web podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas. Neste tutorial, explicamos como você pode usar o WebRTC para configurar conexões peer-to-peer para outros navegadores da web de forma rápida e fácil.**

**Público**

**Este tutorial vai ajudar todos os desenvolvedores que gostariam de aprender como criar aplicativos, como publicidade em tempo real, jogos multiplayer, transmissão ao vivo, e-learning, para citar alguns, onde a ação ocorre em tempo real.**

**Pré-requisitos**

**O WebRTC é uma poderosa ferramenta que pode ser usada para infundir recursos de Comunicações em Tempo Real (RTC) em navegadores e aplicativos móveis. Este tutorial abrange apenas os conceitos básicos da WebRTC e qualquer desenvolvedor regular com algum nível de exposição ao gerenciamento de sessão em tempo real pode facilmente entender os conceitos aqui discutidos.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**WebRTC - Visão geral**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**A Web não é mais estranha à comunicação em tempo real à medida que a WebRTC (Web Real-Time Communication) entra em jogo. Embora tenha sido lançado em maio de 2011, ainda está em desenvolvimento e seus padrões estão mudando. Um conjunto de protocolos é padronizado pela comunicação em tempo real no grupo de trabalho WEB-browsers http://tools.ietf.org/wg/rtcweb/ do IETF (Internet Engineering Task Force) enquanto novos conjuntos de APIs são padronizados pelo Grupo de Trabalho de Comunicações em Tempo Real da Web em http://www.w3.org/2011/04/webrtc/ do W3C (World Wide Web Consortium). Com o surgimento do WebRTC, aplicativos web modernos podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas.**

**Esquema básico**

**O WebRTC permite que você configure conexões peer-to-peer para outros navegadores da Web de forma rápida e fácil. Para criar tal aplicativo a partir do zero, você precisaria de uma grande quantidade de frameworks e bibliotecas que lidam com problemas típicos, como perda de dados, queda de conexão e NAT traversal. Com o WebRTC, tudo isso vem incorporado no navegador fora da caixa. Esta tecnologia não precisa de plugins ou de software de terceiros. É de código aberto e seu código fonte está disponível gratuitamente em http://www.webrtc.org/.**

**A API WebRTC inclui captura de mídia, codificação e decodificação de áudio e vídeo, camada de transporte e gerenciamento de sessão.**

**Captura de mídia**

**O primeiro passo é ter acesso à câmera e ao microfone do dispositivo do usuário. Detectamos o tipo de dispositivos disponíveis, obtenha permissão do usuário para acessar esses dispositivos e gerenciar o fluxo.**

**Codificação e Decodificação de Áudio e Vídeo**

**Não é uma tarefa fácil enviar uma transmissão de dados de áudio e vídeo pela Internet. É aí que a codificação e a decodificação são usadas. Este é o processo de dividir quadros de vídeo e ondas de áudio em pedaços menores e compactá-los. Este algoritmo é chamado de codec. Há uma enorme quantidade de codecs diferentes, que são mantidos por diferentes empresas com diferentes objetivos de negócios. Há também muitos codecs dentro da WebRTC como H.264, iSAC, Opus e VP8. Quando dois navegadores se conectam, eles escolhem o melhor codec suportado entre dois usuários. Felizmente, a WebRTC faz a maior parte da codificação nos bastidores.**

**Camada de transporte**

**A camada de transporte administra a ordem dos pacotes, lida com a perda de pacotes e liga-se a outros usuários. Novamente a WebRTC API nos dá um acesso fácil aos eventos que nos dizem quando há problemas com a conexão.**

**Gerenciamento de Sessão**

**O gerenciamento de sessão trata de gerenciar, abrir e organizar conexões. Isso é comumente chamado de sinalização. Se você transferir fluxos de áudio e vídeo para o usuário, também faz sentido transferir dados colaterais. Isso é feito pela API RTCDataChannel.**

**Engenheiros de empresas como Google, Mozilla, Opera e outros fizeram um ótimo trabalho para trazer essa experiência em tempo real para a Web.**

**Compatibilidade do navegador**

**Os padrões WebRTC são um dos mais rápidos que evoluem na web, portanto, isso não significa que todos os navegadores suportam todos os mesmos recursos ao mesmo tempo. Para verificar se o seu navegador suporta WebRTC ou não, você pode visitar http://caniuse.com/#feat=rtcpeerconnection. Ao longo de todos os tutoriais, recomendo que use o Chrome para todos os exemplos.**

**Tentando WebRTC**

**Vamos começar a usar o WebRTC agora. Navegue no seu navegador para o site de demonstração em https://apprtc.appspot.com/**

**Clique no botão "JOIN". Você deve ver uma notificação suspensa.**

**Clique no botão "Permitir" para começar a transmitir o seu vídeo e áudio para a página da Web. Você deve ver um fluxo de vídeo de si mesmo.**

**Agora abra o URL que você está atualmente em uma nova aba do navegador e clique em "Inscreva-se". Você deve ver dois fluxos de vídeo - um do seu primeiro cliente e outro do segundo.**

**Agora você deve entender porque o WebRTC é uma ferramenta poderosa.**

**Casos de uso**

**A web em tempo real abre a porta para uma nova gama de aplicativos, incluindo bate-papo baseado em texto, tela e compartilhamento de arquivos, jogos, bate-papo por vídeo e muito mais. Além de cOmmunication você pode usar o WebRTC para outros fins como:**

**• marketing em tempo real**

**• publicidade em tempo real**

**• comunicações de back office (CRM, ERP, SCM, FFM)**

**• gerenciamento de RH**

**• redes sociais**

**• serviços de namoro**

**• consultas médicas on-line**

**• serviços financeiros**

**• Vigilância**

**• jogos multiplayer**

**• transmissão ao vivo**

**• e-learning**

**Summary**

**Now você deve ter uma compreensão clara do termo WebRTC. Você também deve ter uma idéia de quais tipos de aplicativos podem ser criados com WebRTC, como você já tentou em seu navegador. Para resumir, a WebRTC é uma tecnologia bastante útil.**

**WebRTC – Arquitetura**

**A arquitetura geral de WebRTC possui um grande nível de complexidade.**

**Aqui você pode encontrar três camadas diferentes –**

**• API para desenvolvedores web - esta camada contém todos os desenvolvedores web da API necessários, incluindo objetos RTCPeerConnection, RTCDataChannel e MediaStrean.**

**• API para fabricantes de navegadores**

**• API Overridable, quais fabricantes de navegador podem se conectar.**

**Componentes de transporte Permitem estabelecer conexões em vários tipos de redes, enquanto os mecanismos de voz e vídeo são responsáveis ​​por transferir fluxos de áudio e vídeo de uma placa de som e câmera para a rede. Para os desenvolvedores da Web, a parte mais importante é a API da WebRTC.**

**Se analisarmos a arquitetura WebRTC do lado do cliente-servidor, podemos ver que um dos modelos mais usados ​​é inspirado no “Trapezoid SIP” (Session Initiation Protocol).**

**Neste modelo, ambos os dispositivos estão executando uma aplicação web de diferentes servidores. O objeto RTCPeerConnection configura fluxos para que eles possam se conectar uns aos outros, peer-to-peer. Esta sinalização é feita via HTTP ou WebSockets.**

**Mas o modelo mais comumente usado é o Triângulo.**

**Neste modelo, ambos os dispositivos usam a mesma aplicação web. Dá ao desenvolvedor web mais flexibilidade ao gerenciar conexões de usuário.**

**A API WebRTC consiste em alguns objetos javascript principais –**

**• RTCPeerConnection**

**• MediaStream**

**• RTCDataChannel**

**O objeto RTCPeerConnection**

**Este objeto é o ponto de entrada principal para a API WebRTC. Isso nos ajuda a conectar-se a pares, inicializar conexões e anexar fluxos de mídia. Ele também gerencia uma conexão UDP com outro usuário.**

**A tarefa principal do objeto RTCPeerConnection é configurar e criar uma conexão entre pares. Podemos facilmente ligar chaves pontos da conexão porque este objeto dispara um conjunto de eventos quando eles aparecem. Esses eventos dão acesso à configuração de nossa conexão.**

**O RTCPeerConnection é um objeto simples de javascript, que você pode simplesmente criar dessa maneira –**

**[código]**

**var conn = new RTCPeerConnection (conf);**

**Conn.onaddstream = function (stream) {**

**// use stream here**

**};**

**[/ Code]**

**O objeto RTCPeerConnection aceita um parâmetro conf, que abordaremos mais adiante nestes tutoriais. O evento onaddstream é disparado quando o usuário remoto adiciona um fluxo de vídeo ou áudio para sua conexão entre pares.**

### **MediaStream API**

**A API *MediaStream***

**Os navegadores modernos oferecem ao desenvolvedor acesso à API getUserMedia, também conhecida como API MediaStream. Existem três pontos-chave de funcionalidade:**

**• Oferece um acesso de desenvolvedor a um objeto de fluxo que representa fluxos de vídeo e áudio**

**• Gerencia a seleção de dispositivos de usuário de entrada no caso de um usuário possuir várias câmeras ou microfones no dispositivo**

**• Fornece uma segurança Nível perguntando ao usuário todo o tempo que ele quer buscar s stream**

**Para testar esta API, vamos criar uma página HTML simples. Ele mostrará um único elemento <video>, pergunte à permissão do usuário para usar a câmera e mostrar uma transmissão ao vivo da câmera na página. Crie um arquivo index.html e adicione –**

**[código]**

**<html>**

**<head>**

**<meta charset = "utf-8">**

**</ head>**

**<body>**

**<video playplay></ video>**

**<script src = "cliente. Js "> </ script>**

**</ body>**

**</ html>**

**[/ code]**

**Em seguida, adicione um arquivo client.js –**

**[code] // verifica se o navegador suporta a função WebRTC hasUserMedia () {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia || Navigator.msGetUserMedia; Retornar !! navigator.getUserMedia; } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia || Navigator.msGetUserMedia; // obtenha os fluxos de vídeo e áudio do navegador da câmera do usuário.getUserMedia ({video: true, audio: true}, função (stream) {var video = document.querySelector ('video'); // insira o fluxo na tag de vídeo Video.src = window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("Erro. ​​WebRTC não é suportado!"); } [/ Code]**

**Agora, abra o index.html e você deve ver o fluxo de vídeo exibindo seu rosto. Mas tenha cuidado, porque o WebRTC funciona apenas do lado do servidor. Se você simplesmente abrir esta página com o navegador, ela não funcionará. Você precisa hospedar esses arquivos nos servidores Apache ou Node ou qual prefere.**

**O objeto RTCDataChannel**

**Enquanto estiver enviando fluxos de mídia entre pares, você também pode enviar dados adicionais usando a API DataChannel. Esta API é tão simples quanto a API MediaStream. O trabalho principal é criar um canal proveniente de um objeto RTCPeerConnection existente**

**- [código] Var peerConn = novo RTCPeerConnection (); // estabelecendo conexão entre pares // ... // fim de estabelecer conexão de pares var dataChannel = peerConnection.createDataChannel ("myChannel", dataChannelOptions); // aqui podemos começar a enviar mensagens diretas para outro peer [/ code]**

**Isto é tudo o que você precisava, apenas duas linhas de código. Tudo o resto é feito na camada interna do navegador. Você pode criar um canal em qualquer conexão entre pares até que o objeto RTCPeerConnection esteja fechado.**

**Summary**

**Você deve ter uma compreensão firme da arquitetura WebRTC. Também cobrimos as APIs MediaStream, RTCPeerConnection e RTCDataChannel. A API WebRTC é um alvo em movimento, portanto, mantenha sempre as últimas especificações**

**Ambiente WebRTC –**

**Para começar a construir nossos aplicativos WebRTC, devemos configurar nosso ambiente de codificação. Em primeiro lugar, você deve ter um editor de texto ou IDE onde você pode editar HTML e Javascript. Há chances de você ter escolhido o preferido porque você está lendo este tutorial. Quanto a mim, estou usando o WebStorm IDE. Você pode baixar sua versão de avaliação em https://www.jetbrains.com/webstorm/. Eu também estou usando o Linux Mint como meu SO de escolha. O outro requisito para aplicativos comuns da WebRTC é ter um servidor para hospedar os arquivos HTML e Javascript. O código não funcionará apenas clicando duas vezes nos arquivos porque o navegador não tem permissão para se conectar a câmeras e microfones, a menos que os arquivos sejam atendidos por um servidor real. Isso é feito, obviamente, devido aos problemas de segurança. Há toneladas de servidores web diferentes, mas neste tutorial, vamos usar o Node.js com o nó estático - • Visite https://nodejs.org/en/ e faça o download A última versão do Node.js. • Descompactá-lo no diretório / usr / local / nodejs. • Abra o arquivo /home/YOUR\_USERNAME/.file de perfil e adicione a seguinte linha ao final - export PATH = $ PATH: / usr / local / Nodejs / bin • Pode reiniciar o computador ou executar a fonte /home/YOUR\_USERNAME/.profile. Agora, o comando do nó deve estar disponível a partir da linha de comando. O comando npm também está disponível. NMP é o gerenciador de pacotes para Node.js. Você pode aprender mais em https://www.npmjs.com/.• Abrir um terminal e executar sudo npm install -g node-static. Isso irá instalar o servidor web estático para Node.js. • Agora navegue até qualquer diretório que contenha os arquivos HTML e execute o comando estático dentro do diretório para iniciar seu servidor da Web. • Você pode navegar para http: // localhost: 8080 para ver Seus arquivos. Há outra maneira de instalar nodejs. Basta executar sudo apt-get install nodejs na janela do terminal. Para testar a instalação do Node.js, abra seu terminal e execute o nodecommand. Digite alguns comandos para verificar como ele funciona –**

**Node.js executa arquivos JavaScript, bem como comandos digitados no terminal. Crie um arquivo index.js com o seguinte conteúdo**

***console.log ("Testing Node.js");***

**em seguida, execute o comando do índice do nó. Você verá o seguinte:**

**ao construir nosso servidor de sinalização ”signaling server”, usaremos uma biblioteca do WebSockets para Node.js. Para instalar em run npm install ws no terminal.**

**Para testar o nosso servidor de sinalização, usaremos o utilitário wscat. Para instalá-lo, execute npm instale -g wscat na sua janela de terminal.**

**Não há protocolos e descrição**

**1 Os protocolos WebRTC**

**Os aplicativos do WebBTC usam UDP (User Datagram Protocol) como o protocolo de transporte. A maioria dos aplicativos da Web hoje são construídos com o uso do protocolo TCP Protocol (Protocolo de Controle de Transmissão)**

**2 Session Description Protocol**

**O SDP é uma parte importante do WebRTC. É um protocolo destinado a descrever as sessões de comunicação de mídia.**

**3 Ao encontrar uma Rota**

**Para se conectar a outro usuário, você deve encontrar um caminho claro em sua própria rede e na rede do outro usuário. Mas há chances de que a rede que você está usando tenha vários níveis de controle de acesso para evitar problemas de segurança.**

**4 Protocolo de transmissão de controle de fluxo**

**Com a conexão entre pares, temos a capacidade de enviar rapidamente dados de vídeo e áudio. O protocolo SCTP é usado hoje para enviar dados blob em cima da nossa conexão de configuração de pares no momento, usando o objeto RTCDataChannel.**

**Summary**

**Neste capítulo, nós abordamos várias das tecnologias que permitem conexões pares, como UDP, TCP, STUN, TURN, ICE , E SCTP. Você deve agora ter uma compreensão ao nível da superfície de como o SDP funciona e seus casos de uso**

.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - MediaStream APIs\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ A API MediaStream foi projetada para facilitar o acesso aos fluxos de mídia de câmeras e microfones locais. O método getUserMedia () é a principal forma de acessar dispositivos de entrada locais. A API possui alguns pontos-chave: • Um fluxo de mídia em tempo real é representado por um objeto de fluxo na forma de vídeo ou áudio • Ele fornece um nível de segurança através de Permissões de usuário perguntando ao usuário antes que um aplicativo da Web possa começar a buscar um fluxo • A seleção de dispositivos de entrada é tratada pela API MediaStream (por exemplo, quando há duas câmeras ou microfones conectados ao dispositivo) Cada objeto MediaStream incluiVários objetos MediaStreamTrack. Eles representam vídeo e áudio de diferentes dispositivos de entrada. Cada objeto MediaStreamTrack pode incluir vários canais (canais de áudio direito e esquerdo). Estas são as partes mais pequenas definidas pela MediaStream API. Existem duas maneiras de produzir objetos MediaStream. Primeiro, podemos renderizar a saída para um elemento de vídeo ou áudio. Em segundo lugar, podemos enviar saída para o objeto RTCPeerConnection, que, em seguida, enviá-lo para um ponto remoto. Usando o MediaStream APILet, crie um aplicativo WebRTC simples. Ele mostrará um elemento de vídeo na tela, pergunte ao usuário permissão para usar a câmera e mostra um fluxo de vídeo ao vivo no navegador. Crie um arquivo index.html - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <video playplay> </ video> < Script src = "client.js"> </ script> </ body> </ html> Em seguida, crie o arquivo client.js e adicione o seguinte: function hasUserMedia () {// verifique se o navegador aceita o retorno da WebRTC !! (Navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e de áudio navigator.getUserMedia ({video: true, audio: true}, function (stream) {var video = document.querySelector ('video'); // inserindo nosso fluxo na tag de vídeo video.src = Window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("WebRTC não é suportado"); } Aqui criamos a função hasUserMedia () que verifica se o WebRTC é suportado ou não. Então, acessamos a função getUserMedia onde o segundo parâmetro é um retorno de chamada que aceita o fluxo que vem do dispositivo do usuário. Em seguida, carregamos nosso fluxo no elemento de vídeo usando window.URL.createObjectURL que cria um URL que representa o objeto dado no parâmetro. Agora atualize sua página, clique em Permitir e você deve ver seu rosto na tela. Lembre-se de executar todos os seus scripts usando o servidor web. Nós já instalamos um no Tutorial de ambiente WebRTC. Medição de APIProperties • MediaStream.active (somente leitura) - Retorna verdadeiro se o MediaStream estiver ativo, ou falso de outra forma. • MediaStream.ended (somente leitura, desativado) - Retornar verdadeiro se o final O evento foi disparado no objeto, o que significa que o fluxo foi completamente lido ou falso se o fim do fluxo não foi alcançado. • MediaStream.id (somente leitura) - Um identificador exclusivo para o objeto. • MediaStream.label (Somente leitura, obsoleta) - Um identificador exclusivo atribuído pelo agente do usuário. Você pode ver como as propriedades acima aparecem no meu navegador - Manipuladores de eventos • MediaStream.onactive - Um manipulador para um evento ativo que é acionado quando um objeto MediaStream se torna ativo • MediaStream.onaddtrack - Um manipulador para um evento addtrack que é disparado quando um novo objeto MediaStreamTrack é adicionado. • MediaStream.onended (obsoleto) - Um manipulador para endedevent que é disparado quando a transmissão está terminando. • MediaStream.oninactive - Um manipulador para um evento inativo que é disparado quando um objeto MediaStream se torna inativo. • MediaStream.onremovetrack - Um manipulador para um evento removetrack que é disparado quando um objeto MediaStreamTrack é removido dele. Métodos • MediaStream.addTrack ( ) - Adiciona o objeto MediaStreamTrack dado como argumento para o MediaStream. Se a faixa já tiver sido adicionada, nada acontece. • MediaStream.clone () - Retorna um clone do objeto MediaStream com uma nova ID. • MediaStream.getAudioTracks () - Retorna uma lista dos objetos MediaStreamTrack de áudio do objeto MediaStream. • MediaStream.getTrackById () - Retorna a faixa por ID. Se o argumento estiver vazio ou o ID não for encontrado, ele retornará nulo. Se você tiver várias faixas com a mesma ID, retorna a primeira. • MediaStream.getTracks () - Retorna uma lista de todos os objetos MediaStreamTrack do objeto MediaStream. • MediaStream.getVideoTracks () - Retorna uma lista dos objetos MediaStreamTrack de video do MediaStream Objeto. • MediaStream.removeTrack () - Remove o MediaStreamTrackobject dado como argumento do MediaStream. Se a faixa já foi removida, nada acontece. Para testar as APIs acima, altere a alteração index.html da seguinte maneira - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf- 8 "/> </ head> <body> <videoplay> </ video> <div> <button id =" btnGetAudioTracks "> getAudioTracks () </ button> </ div> <div> <button id =" btnGetTrackById "> GetTrackById () </ button> </ div> <div> <button id =" btnGetTracks "> getTracks () </ button> </ div> <div> <button id =" btnGetVideoTracks "> getVideoTracks () < / Button> </ div> <div> <button id = "btnRemoveAudioTrack"> removeTrack () - áudio </ button> </ div> <div> <button id = "btnRemoveVideoTrack"> removeTrack () - video </ button > </ Div> <script src = "client.js"> </ script> </ body> </ html> Adicionamos alguns botões para experimentar várias APIs MediaStream. Então, devemos adicionar manipuladores de eventos para o nosso botão recém-criado. Modifique o arquivo client.js dessa maneira - fluxo constante; Função hasUserMedia () {// verifique se o navegador suporta o retorno da WebRTC !! (navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e áudio navigator.getUserMedia ({video: true, audio: true}, função (s) {stream = s; var video = document.querySelector ('video'); // inserindo nosso fluxo no video Etiqueta video.src = window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("WebRTC não é suportado"); } BtnGetAudioTracks.addEventListener ("clique", função () {console.log ("getAudioTracks"); console.log (stream.getAudioTracks ());}); BtnGetTrackById.addEventListener ("clique", função () {console.log ("getTrackById"); console.log (stream.getTrackById (stream.getAudioTracks () [0] .id));}); BtnGetTracks.addEventListener ("clique", função () {console.log ("getTracks ()"); console.log (stream.getTracks ());}); BtnGetVideoTracks.addEventListener ("clique", função () {console.log ("getVideoTracks ()"); console.log (stream.getVideoTracks ());}); btnRemoveAudioTrack.addEventListener ("clique", função () {console .log ("removeAudioTrack ()"); stream.removeTrack (stream.getAudioTracks () [0]);}); BtnRemoveVideoTrack.addEventListener ("clique", função () {console.log ("removeVideoTrack ()"); stream.removeTrack (stream.getVideoTracks () [0]);}); agora atualize sua página. Clique no botão getAudioTracks () e, em seguida, clique no botão removeTrack () - audio. A faixa de áudio agora deve ser removida. Em seguida, faça o mesmo para a faixa de vídeo. Se você clicar no botão getTracks (), você deve ver todos os MediaStreamTracks (todas as entradas de vídeo e áudio conectadas). Em seguida, clique no getTrackById () para obter o áudio MediaStreamTrack. Resumo Neste capítulo, criamos um aplicativo WebRTC simples usando a API MediaStream. Agora, você deve ter uma visão geral clara das várias APIs MediaStream que fazem o WebRTC funcionar .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - API RTCPeerConnection\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ A API RTCPeerConnection é o núcleo da conexão peer-to-peer entre cada um dos navegadores. Para criar os objetos RTCPeerConnection simplesmente writevar pc = RTCPeerConnection (config), onde o argumento config contém pelo menos a chave, iceServers. É uma série de objetos de URL que contém informações sobre os servidores STUN e TURN, usados ​​durante a descoberta dos candidatos do ICE. Você pode encontrar uma lista de servidores STUN públicos disponíveis em code.google.com. Dependendo de se você é o chamador ou o destinatário, o objeto RTCPeerConnection é usado de forma ligeiramente diferente em cada lado da conexão. Aqui é um exemplo do fluxo do usuário - • Registre o manipulador onicecandidate. Ele envia qualquer candidato ICE para o outro parceiro, conforme eles são recebidos. • Registre o manipulador de corrente. Ele controla a exibição do fluxo de vídeo uma vez que é recebido do ponto remoto. • Registre o manipulador de mensagens. Seu servidor de sinalização também deve ter um manipulador para mensagens recebidas do outro ponto. Se a mensagem contiver o objeto RTCSessionDescription, ele deve ser adicionado ao objeto RTCPeerConnection usando o método setRemoteDescription (). Se a mensagem contiver o objeto RTCIceCandidate, ele deve ser adicionado ao objeto RTCPeerConnection usando o método addIceCandidate (). • Use getUserMedia () para configurar seu fluxo de mídia local e adicione-o ao objeto RTCPeerConnection usando o método addStream (). Comece o processo de negociação de oferta / resposta. Este é o único passo em que o fluxo do chamador é diferente do do chamado. O chamador inicia a negociação usando o método createOffer () e registra uma chamada de retorno que recebe o objeto RTCSessionDescription. Em seguida, esse retorno de chamada deve adicionar este objeto RTCSessionDescription ao seu objeto RTCPeerConnection usando setLocalDescription (). E, finalmente, o chamador deve enviar este RTCSessionDescription para o ponto remoto usando o servidor de sinalização. O destinatário, por outro, registra o mesmo retorno de chamada, mas no método createAnswer (). Observe que o fluxo do destinatário é iniciado somente após a oferta ser recebida do chamador.RTCPeerConnection APIProperties • RTCPeerConnection.iceConnectionState (somente leitura) - Retorna um enum RTCIceConnectionState que descreve o estado da conexão. Um evento iceconnectionstatechange é disparado quando esse valor muda. Os valores possíveis - novo - o agente ICE está aguardando candidatos remotos ou coletando verificação de endereços - o agente ICE tem candidatos remotos, mas não encontrou conexão conectada - o agente ICE encontrou uma conexão utilizável, mas ainda está checando Candidato mais remoto para melhor conexão.o completado - o agente ICE encontrou uma conexão utilizável e parou de testar candidatos remotos.o falhou - o agente ICE verificou todos os candidatos remotos, mas não encontrou uma correspondência para pelo menos um componente.o Desconectado - pelo menos um componente não está mais vivo fechado - o agente ICE está próximoD. • RTCPeerConnection.iceGatheringState (somente leitura) - Retorna um enum RTCIceGatheringState que descreve o estado de coleta de ICE para a conexão -o novo - o objeto foi criado apenas. Reunir - o agente ICE está no processo de reunir candidatos para completar o ICE O agente completou a reunião. • RTCPeerConnection.localDescription (somente leitura) - Retorna um RTCSessionDescription descrevendo a sessão local. Pode ser nulo se ainda não tiver sido configurado. • RTCPeerConnection.peerIdentity (somente leitura) - Retorna uma avaliação RTCIdentity. Consiste em um idp (nome de domínio) e um nome que representa a identidade do ponto remoto. • RTCPeerConnection.remoteDescription (somente leitura) - Retorna um RTCSessionDescription descrevendo a sessão remota. Pode ser nulo se ainda não tiver sido configurado. • RTCPeerConnection.signalingState (somente leitura) - Retorna um enumínio RTCSignalingState que descreve o estado de sinalização da conexão local. Este estado descreve a oferta SDP. Um evento signalingstatechange é disparado quando esse valor muda. Os valores possíveis -o estáveis ​​- O estado inicial. Não existe nenhuma oferta de SDP / troca de respostas em andamento; tem-oferta local - o lado local da conexão aplicou localmente uma oferta SDP; tem-oferta remota; o lado remoto da conexão aplicou localmente uma oferta SDP .o have-local-pranswer - uma oferta SDP remota foi aplicada e um prensel SDP aplicado localmente tem-remoção remota - um SDP local foi aplicado e um prensor SDP aplicado remotamente.o fechou - a conexão é Closed.Event HandlersGiven abaixo são os manipuladores de eventos comumente usados ​​de RTCPeerConnection.MethodsGiven abaixo são os métodos comumente usados ​​de RTCPeerConnection.Establishing a ConnectionNow vamos criar um exemplo de aplicação. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor de nó". Serão duas entradas de texto na página, uma para um login e outra para um nome de usuário para o qual queremos conectar. Crie um arquivo index.html e adicione o seguinte código - <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <div> <input type = "text "Id =" loginInput "/> <button id =" loginBtn "> Login </ button> </ div> <div> <input type =" text "id =" otherUsernameInput "/> <button id =" connectToOtherUsernameBtn "> Estabeleça a conexão </ button> </ div> <script src = "client2.js"> </ script> </ body> </ html> Você pode ver que adicionamos a entrada de texto para um login, o botão de login , A entrada de texto para o outro nome de usuário do peer e o botão conectar-para-ele. Agora, crie um arquivo client.js e adicione o seguinte código -var connection = new WebSocket ('ws: // localhost: 9090'); Var name = ""; Var loginInput = document.querySelector ('# loginInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var otherUsernameInput = document.querySelector ('# otherUsernameInput'); Var connectToOtherUsernameBtn = document.querySelector ('# connectToOtherUsernameBtn'); Var connectedUser, myConnection; // quando um usuário clica no botão de login loginBtn.addEventListener ("clique", função (evento) {name = loginInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); // lida com mensagens do servidor connection.onmessage = function (message) {console.log ("Got message", message.data); Var data = JSON.parse (message.data); Switch (data.type) {case "login": onLogin (data.success); pausa; Caso "oferta": onOffer (data.offer, data.name); pausa; Caso "resposta": onAnswer (data.answer); pausa; Caso "candidato": onCandidate (data.candidate); pausa; Padrão: intervalo; }}; // quando um usuário entra na função onLogin (sucesso) {if (success === false) {alert ("oops ... tente um nome de usuário diferente"); } Else {// criando nosso objeto RTCPeerConnection var configuration = {"iceServers": [{"url": "stun: stun.1.google.com: 19302"}]}; MyConnection = new webkitRTCPeerConnection (configuração); Console.log ("o objeto RTCPeerConnection foi criado"); Console.log (myConnection); // configuração de manipulação de gelo // quando o navegador encontra um candidato de gelo, enviamos para outro ponto myConnection.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate }); }}; }}; Connection.onopen = function () {console.log ("Conectado"); }; Connection.onerror = function (err) {console.log ("Got error", err); }; // Alias ​​para enviar mensagens no formato JSON enviar (mensagem) {if (connectedUser) {message.name = connectedUser; } Connection.send (JSON.stringify (mensagem)); }; Você pode ver que estabelecemos uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login, o aplicativo envia seu nome de usuário para o servidor. Se o login for bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o manipulador onicecandidate da configuração que envia todos os icecandidatos encontrados para o outro ponto. Agora abra a página e tente fazer o login. Você deve ver a seguinte saída do console - O próximo passo é criar uma oferta para o outro par. Adicione o seguinte código ao seu arquivo client.js - // configure uma conexão entre pares com outro usuárioConnectToOtherUsernameBtn.addEventListener ("clique", função () {var otherUsername = otherUsernameInput.value; connectedUser = otherUsername; if (otherUsername.length> 0) {// faça uma oferta myConnection.createOffer (function (offer) {console.log ( ); Enviar ({tipo: "oferta", oferta: oferta}); myConnection.setLocalDescription (oferta);}, função (erro) {alerta ("Ocorreu um erro");});}}); // quando alguém quer nos chamar de função onOffer (oferta, nome) {connectedUser = nome; MyConnection.setRemoteDescription (novo RTCSessionDescription (oferta)); MyConnection.createAnswer (função (resposta) {myConnection.setLocalDescription (resposta); send ({type: "answer", answer: answer});}, function (error) {alert ("oops ... error");} ); } // quando outro usuário responde a nossa função de oferta onAnswer (answer) {myConnection.setRemoteDescription (novo RTCSessionDescription (resposta)); // Quando obtivemos um candidato de gelo de outra função de usuário onCandidate (candidato) {myConnection.addIceCandidate (novo RTCIceCandidate (candidato)); } Você pode ver que, quando um usuário clica no botão "Estabelecer conexão", o aplicativo faz uma oferta SDP para o outro par. Nós também definimos onAnswer e onCandidate manipuladores. Recarregue a sua página, abra-a em duas guias, faça o login com dois usuários e tente estabelecer uma conexão entre eles. Você deve ver a seguinte saída do console - Agora a conexão peer-to-peer é estabelecida. Nos próximos tutoriais, adicionaremos fluxos de vídeo e áudio, bem como suporte de bate-papo de texto .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - RTCDataChannel APIs\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Páginas publicitárias\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_WebRTC não só é bom na transferência de transmissões de áudio e vídeo, mas em qualquer dado arbitrário que possamos ter. Este é o lugar onde o objeto RTCDataChannel entra em jogo.RTCDataChannel APIProperties • RTCDataChannel.label (somente leitura) - Retorna uma string contendo o nome do canal de dados. • RTCDataChannel.ordered (somente leitura) - Retorna verdadeiro se a ordem de entrega das mensagens for Garantido ou falso se não for garantido. • RTCDataChannel.protocol (somente leitura) - Retorna uma string contendo o nome do subprotocolo usado para este canal. • RTCDataChannel.id (somente leitura) - Retorna um ID exclusivo para o canal que está configurado no Criação do objeto RTCDataChannel. • RTCDataChannel.readyState (somente leitura) - Retorna o enumínio RTCDataChannelState que representa o estado da conexão. Os possíveis valores -o conexão - Indica que a conexão ainda não está ativa. Este é o estado inicial.o aberto - Indica que a conexão está sendo executada.o fechamento - Indica que a conexão está em processo de desligamento. As mensagens em cache estão em processo de envio ou recebimento, mas nenhuma tarefa recém-criada está aceitando.o fechado - Indica que a conexão não pôde ser estabelecida ou foi encerrada. • RTCDataChannel.bufferedAmount (somente leitura) - Retorna o valor De bytes que foram encaminhados para envio. Esta é a quantidade de dados que ainda não foram enviados através de RTCDataChannel.send (). • RTCDataChannel.bufferedAmountLowThreshold - Retorna o número de bytes em que o RTCDataChannel.bufferedAmount é ocupado como baixo. Quando o RTCDataChannel.bufferedAmount diminui abaixo deste limite, o evento bufferedamountlow é disparado. • RTCDataChannel.binaryType - Retorna o tipo de dados binários transmitidos pela conexão. Pode ser "blob" ou "arraybuffer". • RTCDataChannel.maxPacketLifeType (somente leitura) - Retorna um curto não assinado que indica o comprimento em milissegundos da janela quando a mensagem está no modo não confiável. • RTCDataChannel.maxRetransmits (somente leitura) - Retorna um curto não assinado que indica o número máximo de vezes que um canal irá retransmitir dados se não for entregue. • RTCDataChannel.negotiated (somente leitura) - Retorna um booleano que indica se o canal foi negociado pelo agente do usuário ou por O aplicativo. • RTCDataChannel.reliable (somente leitura) - Retorna um booleano que indica que a conexão pode enviar mensagens em modo não confiável. • RTCDataChannel.stream (somente leitura) - Sinônimo de RTCDataChannel.idEvent Handlers • RTCDataChannel.onopen - Esse manipulador de eventos É chamado quando o evento aberto é disparado. Este evento é enviado quando a conexão de dados foi estabelecida. • RTCDataChannel.onmessage - Este manipulador de eventos é chamado quando o evento da mensagem é disparado. O evento é enviado quando uma mensagem está disponível no canal de dados. • RTCDataChannel.onbufferedamountlow - Este manipulador de eventos é chamado quando o evento bufferedamoutlow é disparado. Este evento é enviado quando RTCDataChannel.bufferedAmount diminui abaixo da propriedade RTCDataChannel.bufferedAmountLowThreshold. • RTCDataChannel.onclose - Este manipulador de eventos é chamado quando o evento de fechamento é disparado. Este evento é enviado quando a conexão de dados foi fechada. • RTCDataChannel.onerror - Este manipulador de eventos é chamado quando o evento de erro é disparado. Este evento é enviado quando um erro tem bEen encontrados. Métodos • RTCDataChannel.close () - Fecha o canal de dados. • RTCDataChannel.send () - Envia os dados no parâmetro através do canal. Os dados podem ser um blob, uma string, um ArrayBuffer ou um ArrayBufferView .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Enviando Mensagens \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Páginas publicitárias\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Nós vamos criar um exemplo simples. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor de nó". Terão três entradas de texto na página, uma para um login, uma para um nome de usuário e outra para a mensagem que queremos enviar Para o outro par. Crie um arquivo index.html e adicione o seguinte código - <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <div> <input type = "text "Id =" loginInput "/> <button id =" loginBtn "> Login </ button> </ div> <div> <input type =" text "id =" otherUsernameInput "/> <button id =" connectToOtherUsernameBtn "> Estabeleça a conexão </ button> </ div> <div> <input type = "text" id = "msgInput" /> <button id = "sendMsgBtn"> Enviar mensagem de texto </ button> </ div> <script src = "Client.js"> </ script> </ body> </ html> Também adicionamos três botões para login, estabelecendo uma conexão e enviando uma mensagem. Agora, crie um arquivo client.js e adicione o seguinte código -var connection = new WebSocket ('ws: // localhost: 9090'); Var name = ""; var loginInput = document.querySelector ('# loginInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var otherUsernameInput = document.querySelector ('# otherUsernameInput'); Var connectToOtherUsernameBtn = document.querySelector ('# connectToOtherUsernameBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var connectedUser, myConnection, dataChannel; // quando um usuário clica no botão de login loginBtn.addEventListener ("clique", função (evento) {name = loginInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); // lida com mensagens do servidor connection.onmessage = function (message) {console.log ("Got message", message.data); Var data = JSON.parse (message.data); Switch (data.type) {case "login": onLogin (data.success); pausa; Caso "oferta": onOffer (data.offer, data.name); pausa; Caso "resposta": onAnswer (data.answer); pausa; Caso "candidato": onCandidate (data.candidate); pausa; Padrão: intervalo; }}; // quando um usuário entra na função onLogin (sucesso) {if (success === false) {alert ("oops ... tente um nome de usuário diferente"); } Else {// criando nosso objeto RTCPeerConnection var configuration = {"iceServers": [{"url": "stun: stun.1.google.com: 19302"}]}; MyConnection = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); Console.log ("o objeto RTCPeerConnection foi criado"); Console.log (myConnection); // configuração de manipulação de gelo // quando o navegador encontra um candidato de gelo, enviamos para outro ponto myConnection.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate }); }}; OpenDataChannel (); }}; Connection.onopen = function () {console.log ("Conectado"); }; Connection.onerror = function (err) {console.log ("Got error", err); }; // Alias ​​para enviar mensagens no formato JSON enviar (mensagem) {if (connectedUser) {message.name = connectedUser; } Connection.send (JSON.stringify (mensagem)); }; Você pode ver que estabelecemos uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login, o aplicativo envia seu nome de usuário para o servidor. Se o login for bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o manipulador onicecandidate da configuração que envia todos os icecandidatos encontrados para o outro ponto. Ele também executa a função openDataChannel () que cria um dataChannel. Observe que ao criar o objeto RTCPeerConnection o segundo argumento no construtor opcional: [{RtpDataChannels: true}] é obrigatório se você estiver usando o Chrome ou o Opera. O próximo passo é criar uma oferta para o outro ponto. Adicione o seguinte código ao seu arquivo client.js - // configure uma conexão peer com outro usuário connectToOtherUsernameBtn.addEventListener ("clique", function () {var otherUsername = otherUsernameInput.value; connectedUser = otherUsername; if (otherUsername.length> 0 ) {// faça uma oferta myConnection.createOffer (função (oferta) {console.log (); enviar ({tipo: "oferta", oferta: oferta}); myConnection.setLocalDescription (oferta);}, função (erro) {Alert ("Ocorreu um erro.");});}}); // quando alguém quer nos chamar de função onOffer (oferta, nome) {connectedUser = nome; MyConnection.setRemoteDescription (novo RTCSessionDescription (oferta)); MyConnection.createAnswer (função (resposta) {myConnection.setLocalDescription (resposta); send ({type: "answer", answer: answer});}, function (error) {alert ("oops ... error");} ); } // quando outro usuário responde a nossa função de oferta onAnswer (answer) {myConnection.setREmoteDescription (novo RTCSessionDescription (resposta)); // Quando obtivemos um candidato de gelo de outra função de usuário onCandidate (candidato) {myConnection.addIceCandidate (novo RTCIceCandidate (candidato)); } Você pode ver que, quando um usuário clica no botão "Estabelecer conexão", o aplicativo faz uma oferta SDP para o outro par. Nós também definimos onAnswer e onCandidate manipuladores. Finalmente, vamos implementar a função openDataChannel () que cria o nosso dataChannel. Adicione o seguinte código ao seu arquivo client.js - // criando função de canal de dados openDataChannel () {var dataChannelOptions = {reliable: true}; DataChannel = myConnection.createDataChannel ("myDataChannel", dataChannelOptions); DataChannel.onerror = function (error) {console.log ("Erro:", erro); }; DataChannel.onmessage = function (event) {console.log ("Got message:", event.data); }; // quando um usuário clicar no botão enviar mensagem sendMsgBtn.addEventListener ("clique", função (evento) {console.log ("enviar mensagem"); var val = msgInput.value; dataChannel.send (val);}) ; Aqui criamos o dataChannel para nossa conexão e adicionamos o manipulador de eventos para o botão "Enviar mensagem". Agora, abra esta página em duas guias, faça o login com dois usuários, estabeleça uma conexão e tente enviar mensagens. Você deve vê-los na saída do console. Observe que o exemplo acima é testado no Opera. Agora você pode ver que o RTCDataChannel é parte extremamente poderosa da API WebRTC. Existem muitos outros casos de uso para este objeto, como jogos peer-to-peer ou compartilhamento de arquivos baseados em torrent .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Sinalização\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Publicidades\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Mais aplicativos WebRTC não são apenas capazes de se comunicar através de vídeo e áudio. Eles precisam de muitos outros recursos. Neste capítulo, vamos construir um servidor de sinalização básica.Signação e Negociação Para conectar-se a outro usuário, você deve saber onde ele está localizado na Web. O endereço IP do seu dispositivo permite que os dispositivos habilitados para Internet enviem dados diretamente entre si. O objeto RTCPeerConnection é responsável por isso. Assim que os dispositivos sabem como se encontrar através da Internet, eles começam a trocar dados sobre quais protocolos e codecs cada dispositivo suporta. Para se comunicar com outro usuário, você simplesmente precisa trocar informações de contato e o resto será feito pela WebRTC. O processo de conexão com o outro usuário também é conhecido como sinalização e negociação. Consiste em algumas etapas: • Crie uma lista de potenciais candidatos para uma conexão entre pares. • O usuário ou um aplicativo seleciona um usuário para estabelecer uma conexão. • A camada de sinalização notifica outro usuário que alguém deseja se conectar a ele. Ele pode aceitar ou recusar. • O primeiro usuário é notificado da aceitação da oferta. • O primeiro usuário inicia o RTCPeerConnection com outro usuário. • Ambos os usuários trocam informações de software e hardware através do servidor de sinalização. • Ambos os usuários trocam informações de localização. A conexão é bem-sucedida ou falha. A especificação WebRTC não contém nenhum padrão sobre a troca de informações. Portanto, tenha em mente que o acima é apenas um exemplo de como a sinalização pode acontecer. Você pode usar qualquer protocolo ou tecnologia que você gosta. Construindo o Servidor O servidor que vamos construir pode conectar dois usuários que não estão localizados no mesmo computador. Criaremos o nosso próprio mecanismo de sinalização. Nosso servidor de sinalização permitirá que um usuário ligue para outro. Uma vez que um usuário tenha chamado outro, o servidor passa a oferta, responde, os candidatos do ICE entre eles e configura uma conexão WebRTC. O diagrama acima é o fluxo de mensagens entre usuários ao usar o servidor de sinalização. Em primeiro lugar, cada usuário se registra com o servidor. No nosso caso, este será um nome de usuário de string simples. Uma vez que os usuários se registraram, eles podem ligar-se. O usuário 1 faz uma oferta com o identificador de usuário que deseja chamar. O outro usuário deve responder. Finalmente, os candidatos do ICE são enviados entre os usuários até que eles possam fazer uma conexão. Para criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de pares WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já instalamosN criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta ao usuário conectado dizendo "Olá do servidor". Agora, o servidor de nó de execução e o servidor devem começar a ouvir conexões de soquete. Para testar nosso servidor, usaremos o utilitário wscat que também já instalamos. Esta ferramenta ajuda a se conectar diretamente ao servidor WebSocket e a testar os comandos. Execute nosso servidor em uma janela de terminal, abra outro e execute o comando wscat -c ws: // localhost: 9090. Você deve ver o seguinte no lado do cliente - O servidor também deve registrar o usuário conectado - Registro do usuárioNo nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão, então nós sabemos onde enviar mensagens. Vamos mudar o nosso manual de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // tudo conectado ao servidor usersvar users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário enviar uma mensagem com o tipo de login, nós - • Verifique se alguém já fez logon com este nome de usuário • Se assim for , Então, diga ao usuário que ele não iniciou sessão com êxito • Se ninguém estiver usando esse nome de usuário, nós nos adicionamos Ername como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } A função acima garante que todas as nossas mensagens são enviadas no formato JSON. Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); agora vamos testar nosso servidor com o comando de login. Tenha em mente que todas as mensagens devem ser codificadas no formato JSON. Execute nosso servidor e tente fazer o login. Você deve ver algo como isto - Fazendo uma ligação. Após o login bem-sucedido, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Respondendo A resposta à resposta tem um padrão similar que usamos no negociador de ofertas. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após a oferta hander -case "answer": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; Você pode ver como isso é semelhante ao manipulador da oferta. Observe que este código segue as funções createOffer e createAnswer no objeto RTCPeerConnection. Agora, podemos testar nosso mecanismo de oferta / resposta. Conecte dois clientes ao mesmo tempo e tente oferecer e responder. Você deve ver o seguinte - Neste exemplo, oferecer e responder são strings simples, mas em um aplicativo real serão preenchidas com tEle SDP data.ICE Candidatos A parte final está lidando com candidato ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Ele deve funcionar de forma semelhante aos manipuladores de oferta e resposta. Como ativar a conexão Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .logname, {type: "leave"});}}}}}); agora, se a A conexão termina, nossos usuários serão desconectados. O fechamento será disparado quando um usuário fechar a janela do navegador enquanto ainda estamos em oferta, resposta ou estado candidato. O Servidor de Sinalização CompletoHere é o código inteiro do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws' ).Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe detalhes da oferta var conn = users [data.name]; if (conn ! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Então o trabalho está pronto eo nosso servidor de sinalização está pronto. Lembre-se de que fazer coisas fora de ordem ao fazer uma conexão WebRTC pode causar problemas.SummaryNeste capítulo, nós construímos um servidor de sinalização simples e direto. Atravessamos o processo de sinalização, registro de usuário e mecanismo de oferta / resposta. Também implementamos o envio de candidatos entre usuários .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Suporte para navegador \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ADvertisements\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ A Web está se movendo tão rápido e está sempre melhorando. Novos padrões são criados todos os dias. Os navegadores permitem que as atualizações sejam instaladas sem o usuário saber, então você deve manter-se atualizado com o que está acontecendo no mundo da Web e da WebRTC. Aqui está uma visão geral sobre o que isso faz hoje. Suporte do navegador. Todo o navegador não possui todos os mesmos recursos do WebRTC ao mesmo tempo. Navegadores diferentes podem estar à frente da curva, o que faz com que alguns recursos WebRTC funcionem em um navegador e não outro. O suporte atual para WebRTC no navegador é mostrado na seguinte imagem. Você pode verificar um status de suporte atualizado do WebRTC em http://caniuse.com/#feat=rtcpeerconnection.Chrome, Firefox e Opera. As últimas versões do Chrome, Firefox e Opera nos sistemas operacionais convencionais do PC, como Mac OS X, Windows e Linux, todos apoiam a WebRTC out-of-the-box. E, o mais importante, os engenheiros das equipes de desenvolvedores do Chrome e do Firefox estão trabalhando juntos para corrigir problemas para que esses dois navegadores possam se comunicar facilmente. Os sistemas operacionais Android OSOn Android, os aplicativos WebRTC para o Chrome e o Firefox devem funcionar de forma rápida. Eles são capazes de trabalhar com outros navegadores após a versão Android Ice Cream Sandwich (4.0). Isto é devido ao compartilhamento de código entre versões de desktop e móveis. O AppleApple ainda não fez nenhum anúncio sobre seus planos para suportar o WebRTC no Safari no OS X. Uma das soluções possíveis para aplicações iOS nativas híbridas os para incorporar o código WebRTC diretamente O aplicativo e carregar este aplicativo em um WebView.Internet ExplorerMicrosoft não suporta WebRTC em desktops. Mas eles confirmaram oficialmente que vão implementar ORTC (Object Realtime Communications) em futuras versões do IE (Edge). Eles não estão planejando suportar o WebRTC 1.0. Eles rotularam sua ORTC como WebRTC 1.1, embora seja apenas um aprimoramento da comunidade e não o padrão oficial. Recentemente, eles adicionaram o suporte ORTC à versão mais recente do Microsoft Edge. Você pode aprender mais em https://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoftedge/.SummaryNotice que o WebRTC é uma coleção de APIs e protocolos, não Uma única API. O suporte para cada um deles está se desenvolvendo em diferentes navegadores e sistemas operacionais em um nível diferente. Uma ótima maneira de verificar o nível mais recente de suporte é através do http://canisue.com. Ele acompanha a adoção de APIs modernas em vários navegadores. Você também pode encontrar as informações mais recentes sobre os suportes do navegador, bem como demonstrações WebRTC em http://www.webrtc.org, que é suportado por Mozilla, Google e Opera .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Suporte móvel\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Páginas publicitárias\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ No mundo móvel, o O suporte da WebRTC não está no mesmo nível que está nos desktops. Os dispositivos móveis têm sua própria maneira, então a WebRTC também é algo diferente nas plataformas móveis. Ao desenvolver um aplicativo WebRTC para área de trabalho, consideramos o uso do Chrome, Firefox ou Opera. Todos eles suportam o WebRTC fora da caixa. Em geral, você só precisa de um navegador e não se preocupe com o hardware da área de trabalho. No mundo móvel, existem três modos possíveis para o WebRTC hoje - • O aplicativo nativo • O aplicativo do navegador • O navegador nativoAndroidIn 2013, o navegador Firefox para Android foi Apresentou o suporte da WebRTC fora da caixa. Agora você pode fazer chamadas de vídeo em dispositivos Android usando o navegador móvel do Firefox. Ele possui três componentes principais do WebRTC - • PeerConnection - permite chamadas entre navegadores • getUserMedia - fornece acesso à câmera e ao microfone • DataChannels - fornece transferência de dados peer-to-peerGoogle O Chrome para Android também oferece suporte para WebRTC. Como você já notou, as características mais interessantes geralmente aparecem no Chrome. No ano passado, o navegador móvel da Opera apareceu com o suporte da WebRTC. Então, para o Android, você tem Chrome, Firefox e Opera. Outros navegadores não suportam WebRTC.iOS Infelizmente, o WebRTC não é suportado no iOS agora. Embora o WebRTC funcione bem no Mac ao usar o Firefox, o Opera ou o Chrome, ele não é suportado no iOS. Hoje em dia, seu aplicativo WebRTC não funcionará nos dispositivos móveis da Apple fora da caixa. Mas há um navegador - Bowser. É um navegador web desenvolvido pela Ericsson e ele suporta o WebRTC fora da caixa. Você pode verificar sua página inicial em http://www.openwebrtc.org/bowser/.Today, é a única maneira amigável de suportar seu aplicativo WebRTC no iOS. Outra maneira é desenvolver um aplicativo nativo sozinho. Windows PhonesMicrosoft não suporta WebRTC em plataformas móveis. Mas eles confirmaram oficialmente que vão implementar ORTC (Object Realtime Communications) em futuras versões do IE. Eles não estão planejando suportar o WebRTC 1.0. Eles rotularam sua ORTC como WebRTC 1.1, embora seja apenas um aprimoramento da comunidade e não o padrão oficial. Hoje, os usuários do Windows Phone não podem usar aplicativos WebRTC e não há como vencer esta situação. Os aplicativos do BlackberryWebRTC também não são suportados no Blackberry. Usando um navegador nativo WebRTCO caso mais conveniente e confortável para os usuários utilizar o WebRTC está usando o navegador nativo do dispositivo. Nesse caso, o dispositivo está pronto para trabalhar com configurações adicionais. Hoje em dia, apenas os dispositivos Android que são a versão 4 ou superior fornecem esse recurso. A Apple ainda não mostra nenhuma atividade com o suporte da WebRTC. Portanto, os usuários do Safari não podem usar aplicativos WebRTC. A Microsoft também não apresentou no Windows Phone 8. Usando a WebRTC por meio de aplicativos do navegador. Isso significa usar aplicativos de terceiros (navegadores da Web não-nativos) para fornecer os recursos do WebRTC. Por enquanto, existem dois desses aplicativos de terceiros. Bowser, que é a única maneira de trazer os recursos do WebRTC para o dispositivo iOS e o Opera, que é uma ótima alternativa para a plataforma Android. O resto dos navegadores móveis disponíveis não suporta WebRTC. Aplicativos móveis móveis Como você pode ver, a WebRTC ainda não possui um grande suporte no mundo móvel. Assim, uma das soluções possíveis é desenvolver aplicações nativas que utilizem a API WebRTC. Mas não é a melhor escolha porque o principal recurso WebRTC é uma solução multi-plataforma. De qualquer forma, em alguns casos, esta é a única maneira porque um aplicativo nativo pode utilizar funções ou recursos específicos do dispositivo que não são suportados pelos navegadores HTML5. Conexão de fluxo de vídeo para dispositivos móveis e de mesa O primeiro parâmetro da API getUserMedia espera um objeto de chaves e Valores indicando ao navegador como processar fluxos. Você pode verificar o conjunto completo de restrições em https://tools.ietf.org/html/draft-alvestrand-constraints-resolution-03. Você pode configurar a ração de aspecto de vídeo, frameRate e outros parâmetros opcionais. Suportar dispositivos móveis é uma das maiores dores, porque os dispositivos móveis possuem espaço de tela limitado e recursos limitados. Você pode querer que o dispositivo móvel capture apenas uma resolução de resolução de 2 x 360 ou fluxo de vídeo menor para economizar energia e largura de banda. Usar a string do agente do usuário no navegador é uma boa maneira de testar se o usuário está em um dispositivo móvel ou não. Vamos ver um exemplo. Crie o arquivo index.html - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <videoplay video> </ video> < Script src = "client.js"> </ script> </ body> </ html> Em seguida, crie o seguinte arquivo client.js - // restrições para o navegador de desktop var desktopConstraints = {video: {mandatory: {maxWidth: 800, MaxHight: 600}}, áudio: true}; // restrições para o navegador móvel var mobileConstraints = {video: {obrigatório: {maxWidth: 480, maxHeight: 320,}}, audio: true} // se um usuário estiver usando um navegador móvel se (/ Android | iPhone | iPad / I.test (navigator.userAgent)) {var constraints = mobileConstraints; } Else {var constraints = desktopConstraints; } Função hasUserMedia () {// verifique se o navegador suporta o retorno da WebRTC! (Navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e de áudio navigator.getUserMedia (restrições, função (fluxo) {var video = document.querySelector ('video'); // inserindo nosso fluxo na tag de vídeo video.src = window.URL.createObjectURL (fluxo );}, Function (err) {}); } Else {alert ("WebRTC não é suportado"); } Execute o servidor web usando o comando estático e abra a página. Você deve ver que é 800x600. Em seguida, abra esta página em uma viewport móvel usando ferramentas cromadas e verifique a resolução. Deve ser de 20%. As restrições são a maneira mais fácil de aumentar o desempenho do seu aplicativo WebRTC.SummaryNeste capítulo, aprendemos sobre os problemas que podem ocorrer ao desenvolver aplicativos WebRTC para dispositivos móveis. Descobrimos diferentes limitações de suporte à API WebRTC em plataformas móveis. Também lançamos um aplicativo de demonstração onde estabelecemos restrições diferentes para navegadores de desktop e móveis .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Demo de Vídeo \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Publicidades\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página AnteriorPágina seguinte \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Neste capítulo, vamos construir um aplicativo cliente que permita que dois usuários em dispositivos separados se comuniquem usando o WebRTC. Nossa aplicação terá duas páginas. Um para o login e outro para chamar outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples. Servidor de Sinalização Para criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. É aqui que vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso manipulador de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); Se o usuário enviar uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário • Em caso afirmativo, diga ao usuário que ele não tenha logado com sucesso • Se ninguém estiver usando esse nome de usuário, nós Adicione o nome de usuário como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); Após o login com êxito, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectassem de outro usuário, devemos implementarA função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é o código completo do nosso servidor de sinalização - // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; if (conn! = Null) {// configuração que UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Aplicação de cliente. Uma maneira de testar esta aplicação está abrindo duas guias do navegador e tentando ligar uma. Por fim, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da web. Você pode aprender mais em http://getbootstrap.com/. Crie uma pasta chamada, por exemplo, "videochat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um arquivo package.json (é necessário gerenciar dependências npm) e adicione o seguinte - {"name": "webrtc-videochat", "versão": "0.1.0", "description": "webrtc -videochat "," autor ":" Autor "," licença ":" BSD-2-Clause "} Em seguida, execute o npm install bootstrap. Isso instalará a biblioteca bootstrap na pasta videochat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Video Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } Video {background: black; Borda: 1px sólido cinza; } .call-page {position: relative; Exibição: bloco; Margem: 0 auto; Largura: 500px; Altura: 500px; } #localVideo {width: 150px; Altura: 150px; Posição: absoluta; Topo: 15px; Direito: 15px; }#remoteVideo {width: 500px; Altura: 500px; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Video Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" c lass = "form-control formgroup" placeholder = "Login" requerido = "" autofocus = ""> <botão id = "loginBtn" class = "btn btn-lg btn-primary btnblock"> Inscreva-se </ button> </ div> </ div> </ div> <Div id = "callPage" class = "call-page"> <video id = "localVideo" reprodução automática> </ video> <video id = "remoteVideo" reprodução automática> </ video> <div class = "row text-center "<Div class =" col-md-12 "> <input id =" callToUsernameInput "type =" text "placeholder =" nome de usuário para ligar "/> <button id =" callBtn "class =" btn-successful btn " > Ligar </ button> <button id = "hangUpBtn" class = "btn-danger btn"> Desligar </ button> </ div> </ div> </ div> <script src = "client.js"> </ Script> </ body> </ html> Esta página deve ser familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Você deve ver os dois elementos de vídeo para fluxos de vídeo locais e remotos. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); // esconde a página de chamada callPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {// exibe a página de chamada se o login for bem-sucedido loginPage.style.display = "none"; CallPage.style.display = "block"; // start peer connection}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem-sucedido, mostramos a página de chamada e começamos a configurar uma conexão entre pares. Para iniciar uma conexão entre pares, precisamos - • Obtenha um fluxo da câmera web. • Crie o objeto RTCPeerConnection. Adicione o seguinte código ao " Seletores UI bloqueiam "-var localVideo = document.querySelector ('# localVideo'); Var remoteVideo = document.querySelector ('# remoteVideo'); Var yourConn; Var stream; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de vídeo local navigator.webkitGetUserMedia ({video: true, audio: true}, function (myStream) {stream = myStream; // exibindo fluxo de vídeo local na página localVideo.src = window.URL. CreateObjectURL (stream); // usando GoogleServidor público stun var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração); // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão de pares, mostramos o seu.Conn.onaddstream = função (e) {remoteVideo.src = window.URL.createObjectURL (e.stream); }; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; }, Função (erro) {console.log (erro); }); }}; Agora, se você executar o código, a página deve permitir que você faça login e exiba seu fluxo de vídeo local na página. Agora estamos prontos para iniciar uma chamada. Em primeiro lugar, enviamos uma oferta para outro usuário. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando uma chamada callBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // Crie uma oferta yourConn.createOffer (função (oferta) {enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta" );});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remota, lida com respostaAnswer (resposta) {yourConn.setRemoteDescription (novo RTCSessionDescription (answer));}; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }: Adicionamos um manipulador de cliques ao botão Ligar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador onmessage. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O último passo é implementar o recurso de desligamento. Isso deixará de transmitir dados e informará ao outro usuário para fechar a chamada. Adicione o seguinte código - // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteVideo.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null; } Quando o usuário clicar no botão Desligar - • Ele enviará uma mensagem "deixar" para o outro usuário • Ele fechará o RTCPeerConnection e destruirá a conexão localNada execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. Você pode então chamar a guia e desligar a chamada. O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se a nossa sinalização servervar conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var localVideo = document.querySelector ('# localVideo'); Var remoteVideo = document.querySelector ('# remoteVideo'); Var yourConn; Fluxo de var; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({ Digite: "login", nome: nome});}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // StartiNg uma conexão de pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // recebendo fluxo de vídeo local navigator.webkitGetUserMedia ({video: true, audio: true}, função ( MyStream) {stream = myStream; // exibindo fluxo de vídeo local na página localVideo.src = window.URL.createObjectURL (stream); // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "Stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração); // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona fluxo para a conexão entre pares, Nós exibimos o seu nome.Con.onaddstream = função (e) {remoteVideo.src = window.URL.createObjectURL (e.stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) { Enviar {{type: "candidato", candidato: event.candidate});}};}; função (erro) {console.log (erro);}); }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função remota do usuário, lidar com resposta (resposta) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteVideo.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null; } Resumo Esta demonstração fornece uma linha de base de recursos que cada aplicativo WebRTC precisa. Para melhorar esta demo, você pode adicionar a identificação do usuário através de plataformas como o Facebook ou o Google, lidar com a entrada do usuário para dados inválidos. Além disso, a conexão WebRTC pode falhar devido a várias razões, como não suportar a tecnologia ou não poder atravessar firewalls. Um valor de trabalho passou a tornar o aplicativo WebRTC está estável .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Demonstração de Voz \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Publicidades\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página AnteriorPágina seguinte \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Neste capítulo, vamos construir um aplicativo cliente que permita que dois usuários em dispositivos separados se comuniquem usando fluxos de áudio WebRTC. Nossa aplicação terá duas páginas. Um para login e outro para fazer uma ligação de áudio para outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples.Signaling ServerPara criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso manual de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Altere o topo do nosso arquivo - // exige nossa biblioteca do websocket var WebSocketServer = requirE ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário envia uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário. • Se Então, diga ao usuário que ele não iniciou sessão com sucesso. • Se ninguém estiver usando esse nome de usuário, adicionamos usernam E como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão-connection.on ("fechar", função () {if (connection.name) {excluir usuários [connection.name];}}); Após o login bem-sucedido, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é O código completo do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor Obtém uma mensagem de uma conexão de usuário conectada.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};} // tipo de comutação do usuário euSsage switch (data.type) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário logado", data.name); // se alguém estiver logado com este nome de usuário, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false}); } Else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; Connection.name = data.name; SendTo (conexão, {digite: "login", sucesso: true}); } pausa; Caso "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Aplicação de cliente. Uma maneira de testar este aplicativo está abrindo duas guias do navegador e tentando fazer uma ligação de áudio entre si. Em primeiro lugar, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da web. Você pode aprender mais em http://getbootstrap.com/. Crie uma pasta chamada, por exemplo, "audiochat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um pacote de arquivos.json (é necessário gerenciar dependências npm) e adicione o seguinte - {"nome": "webrtc-audiochat", "versão": "0.1.0", "descrição": "webrtc -audiochat "," autor ":" Autor "," licença ":" BSD-2-Clause "} Em seguida, execute o npm install bootstrap. Isso instalará a biblioteca bootstrap na pasta audiochat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Voice Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Voice Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <button id =" loginBtn "class =" btn btn-lg btn-primary btnblock "> Iniciar sessão </ button> </ div> </ div> </ div> < Div id = "callPage" class = "call-page"> <div class = "row"> <div class = "col-md-6 text-right"> áudio local: <audio id = "localAudio" controla a reprodução automática> </ Audio> </ div> <div class = "col-md-6 text-left"> Áudio remoto: <audio id = "remoteAudio" controla a reprodução automática> </ audio> </ div> </ div> <div Class = "row text-center"> <div class = "col-md-12"> <input id = "callToUsernameInput" type = "text" placeholder = "nome de usuário para ligar" /> <button id = classe "callBtn" = "Btn-sucesso btn"> Ligar </ button> <botão id = "hangUpBtn" class = "btn-danger btn"> Desligar </ button> </ div> </ div> </ div> <script src = "Client.js"> </ script> </ body> </ html> Esta página deve ser b É familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Você deve ver os dois elementos de áudio para os fluxos de áudio locais e remotos. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {contrasOle.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem sucedido, mostramos a página de chamada e começamos a configurar uma conexão de pares. Para iniciar uma conexão entre pares, precisamos - • Obter um fluxo de áudio de um microfone • Criar o objeto RTCPeerConnectionAdicione o seguinte código ao "bloco de seleção UI "-var localAudio = document.querySelector ('# localAudio'); Var remoteAudio = document.querySelector ('# remoteAudio'); Var yourConn; Var stream; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de áudio local navigator.webkitGetUserMedia ({video: falso, áudio: true}, função (myStream) {stream = myStream; // exibindo fluxo de áudio local na página localAudio.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteAudio.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate",});}};}, function (error) {console. Log (erro);}); }}; Agora, se você executar o código, a página deve permitir que você faça login e exiba seu fluxo de áudio local na página. Agora estamos prontos para iniciar uma chamada. Em primeiro lugar, enviamos uma oferta para outro usuário. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando uma chamada callBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // Crie uma oferta yourConn.createOffer (função (oferta) {enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta" );});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("ERror ao criar uma resposta ");});}; // quando obtivemos uma resposta a partir de uma função remota do usuário, toque a resposta (resposta) {yourConn.setRemoteDescription (nova RTCSessionDescription (resposta));}; // quando conseguimos um gelo Candidato de uma função de usuário remoto handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato));}; Adicionamos um manipulador de cliques ao botão Chamar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador de mensagens. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O último passo é implementar o recurso de desligamento. Isso irá parar de transmitir dados e dizer ao outro usuário para fechar a chamada. Adicione o seguinte código - // hang up hangUpBtn .addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); function handleLeave () {connectedUser = null; remoteAudio.src = null; yourConn.close (); YourConn.onicecandidate = null; yourConn.onaddstream = null;}; Quando o usuário clicar no Hang Up Botão - • Ele enviará uma mensagem de "licença" para o outro usuário • Ele fechará o RTCPeerConnection e destruirá a conexão localmente. Execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. Você pode então fazer uma ligação de áudio para a guia e desligar a chamada. O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var localAudio = document.querySelector ('# localAudio'); Var remoteAudio = document.querySelector ('# remoteAudio'); Var yourConn; Fluxo de var; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de áudio local navigator.webkitGetUserMedia ({video: falso, áudio: true}, função (myStream) {stream = myStream; // exibindo fluxo de áudio local na página localAudio.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteAudio.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate});}};}}, função ( Erro) {console.log (erro);}); }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("ErrorAo criar uma resposta ");});}; // quando obtivemos uma resposta a partir de uma função remota do usuário, responda a resposta (resposta) {yourConn.setRemoteDescription (nova RTCSessionDescription (resposta));}; // quando obtivemos um candidato ao gelo A partir de uma função remota de usuário handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato));}; hang hang uphangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; remoteAudio.src = null; yourConn.close (); yourConn.onicecandidate = null; yourConn.onaddstream = null;}; \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Demonstração de texto \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Páginas publicitárias\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Neste Capítulo, vamos construir um aplicativo cliente que permita que dois usuários em dispositivos separados enviem mensagens entre si usando WebRTC. Nosso aplicativo terá duas páginas. Um para login e D o outro para enviar mensagens para outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples.Signaling ServerPara criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso manual de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário envia uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário. • Se Então, diga ao usuário que ele não iniciou sessão com sucesso. • Se ninguém estiver usando esse nome de usuário, adicionamos userna Eu como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); Após o login com êxito, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se UserB existir, então seE ele oferece detalhes var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); Em primeiro lugar, temos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão entre pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é O código completo do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; if (conn! = Null) {/ / Configuração que UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Aplicação de cliente. Uma maneira de testar este aplicativo é abrir duas guias de navegador e tentar enviar uma mensagem. Primeiro, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da web. Você pode aprender mais em http://getbootstrap.com/.Crie uma pasta chamada, por exemplo, "textchat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um pacote de arquivos.json (é necessário gerenciar dependências de npm) e adicione o seguinte - {"nome": "webrtc-textochat", "versão": "0.1.0", "descrição": "webrtc -textchat "," autor ":" Autor "," licença ":" BSD-2-Clause "} Em seguida, execute npm install bootstrap. Isso instalará a biblioteca bootstrap na pasta textchat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Text Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Text Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <button id =" loginBtn "class =" btn btn-lg btn-primary btnblock "> Iniciar sessão </ button> </ div> </ div> </ div> < Div id = "callPage" class = "container de página de chamada"> <div class = "row"> <div class = "col-md-4 col-md-offset-4 text-center"> <div class = " Painel de painel-primário "> <div class =" painel-título "> bate-papo de texto </ div> <div id =" chatarea "class =" painel-corpo texto-esquerdo "> </ div> </ div> </ Div> </ div> <div class = "linha de texto do centro do formulário-grupo"> <div class = "col-md-12"> <input id = "callToUsernameInput" type = "text" placeholder = "nome de usuário para ligar "/> <Botão id =" callBtn "class =" btn-sucesso btn "> Chamar </ button> <botão id =" hangUpBtn "class =" btn-danger btn "> Desligar </ button> </ div> </ Div> <div class = "row text-center"> <div class = "col-md-12"> <input id = "msgInput" type = "Texto" espaço reservado = "mensagem" /> <botão id = "sendMsgBtn" class = "btn-sucesso btn"> Enviar </ button> </ div> </ div> </ div> <script src = "cliente. Js "> </ script> </ body> </ html> Esta página deve ser familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Na página "chat", você deve ver a etiqueta div com a identificação "chatarea" onde todas as nossas mensagens serão exibidas. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); Var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", Nome: nome}); }}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem-sucedido, mostramos a página de chamada, configuramos uma conexão entre pares e criamos um canal de dados. Para iniciar uma conexão entre pares com um canal de dados que precisamos - • Criar o objeto RTCPeerConnection • Criar um canal de dados dentro do nosso objeto RTCPeerConnectionAdd O seguinte código para o "bloco de seletores UI" -var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Var dataChannel; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; Se o login foi bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o gerenciador onicecandidate da configuração, que envia todos os icecandidatos encontrados para o outro ponto. Ele também cria um dataChannel. Observe que ao criar o objeto RTCPeerConnection o segundo argumento no construtor opcional: [{RtpDataChannels: true}] é obrigatório se você estiver usando o Chrome ou o Opera. O próximo passo é criar uma oferta para o outro ponto. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando um callcallBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // create Uma oferta yourConn.createOffer (função (oferta) {enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta") ;});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");});}; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }: Adicionamos um manipulador de cliques ao botão Ligar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador onmessage. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O próximo passo é implementar o recurso de desligamento. Isso deixará de transmitir dados e informará ao outro usuário para fechar o canal de dados. Adicione o seguinte código - // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; } Quando o usuário clicar no botão Desligar - • Ele enviará uma mensagem "deixar" para o outro usuário. • Ele fechará o RTCPeerConnection e também o canal de dados. O último passo é enviar uma mensagem para outro ponto. Adicione o manipulador "clique" ao botão "enviar mensagem" - // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = Nome + ":" + val + "<br />"; // enviando uma mensagem para um dado de dados de pares conectadosChannel.send (val); msgInput.value = "";}); agora execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. VocêPode então configurar uma conexão entre pares para o outro usuário e enviar-lhe uma mensagem, bem como fechar o canal de dados clicando no botão "Desligar". O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização");}; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Dados varCanal; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; }; // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = nome + ":" + val + "<br />" ; // enviando uma mensagem para um número de usuário compartilhado por telefoneChannel.send (val); msgInput.value = "";}); \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Segurança\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Publicidades\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página anteriorPróxima página \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Neste capítulo, vamos adicionar recursos de segurança ao servidor de sinalização que criamos Na "WebRTC Signaling "capítulo. Haverá dois aprimoramentos - • Autenticação de usuário usando o banco de dados Redis • Ativando a conexão de soquete seguroFirstly, você deve instalar o Redis. • Baixe a versão estável mais recente em http://redis.io/download(3.05 no meu caso) • Descompactar • Interior A pasta de download executada sudo make install • Após a conclusão da instalação, execute make test para verificar se tudo está funcionando corretamente. Redis possui dois comandos executáveis ​​- • redis-cli - interface de linha de comando para Redis (parte do cliente) • redis-server - Redis data storePara executar o redis server type redis-server no console do terminal. Você deve ver o seguinte: agora abra uma nova janela do terminal e execute o redis-cli para abrir uma aplicação cliente. Basicamente, a Redis é uma base de dados de valores-chave. Para criar uma chave com um valor de seqüência de caracteres, você deve usar o comando SET. Para ler o valor da chave, você deve usar o comando GET. Vamos adicionar dois usuários e senhas para eles. As chaves serão os nomes de usuário e os valores dessas chaves serão as senhas correspondentes. Agora, devemos modificar nosso servidor de sinalização para adicionar uma autenticação de usuário. Adicione o seguinte código à parte superior do arquivo server.js - // requer a biblioteca redis no Node.js var redis = require ("redis"); // criando o objeto redis client var redisClient = redis.createClient (); no código acima, exigimos a biblioteca Redis para Node.js e criamos um cliente redis para o nosso servidor. Para adicionar a autenticação, modifique o manipulador de mensagens na conexão Objeto - // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on (' Mensagem ', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // verifique se um usuário é autenticado se (data.type! = "Login") {// se o usuário não for autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", mensagem: "You Não são autenticados "}); return;}} // tipo de comutação da mudança de mensagem do usuário (data.type) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", dados .name); // obter senha para este nome de usuário de re Esta base de dados redisClient.get (data.name, function (err, reply) {// verifique se a senha coincide com a armazenada em redis var loginSuccess = reply === data.password; // se alguém estiver logado com esse nome de usuário ou senha incorreta, recuse se (usuários [data.name] ||! LoginSuccess) {sendTo (connection, {type: "login", success: false}); } Else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; Connection.name = data.name; Connection.isAuth = true; SendTo (conexão, {digite: "login", sucesso: true}); }}); pausa; }}); } // ... // \*\*\*\*\* outros manipuladores \*\*\*\*\*\*\* No código acima, se um usuário tentar entrar, obtemos da Redis sua senha, verifique se ele coincide com o armazenado, e se É bem sucedido que nós armazenamos seu nome de usuário no servidor. Também adicionamos a bandeira isAuth à conexão para verificar se o usuário está autenticado. Observe este código - // verifique se um usuário é autenticado se (data.type! = "Login") {// se o usuário não está autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", Mensagem: "Você não está autenticado"}); Retorna; }} Se um usuário não autenticado tentar enviar uma oferta ou deixar a conexão, simplesmente enviamos um erro para trás. O próximo passo é habilitar uma conexão de soquete segura. É altamente recomendado para aplicações WebRTC. PKI (Public Key Infrastructure) é uma assinatura digital de uma autoridade de certificação (Certificate Authority). Os usuários então verificam se a chave privada usada para assinar um certificado corresponde à chave pública do certificado da CA. Para fins de desenvolvimento. Nós usaremos um certificado de segurança auto-assinado. Usamos o openssl. É uma ferramenta de código aberto que implementa protocolos SSL (Secure Sockets Layer) e TLS (Transport Layer Security). Geralmente é instalado por padrão em sistemas Unix. Execute a versão openssl -a para verificar se ela está instalada. Para gerar chaves de certificado de segurança públicas e privadas, você deve seguir as etapas abaixo: • Gerar uma senha temporária do servidor keyopenssl genrsa -des3 -passout pass: x -out server.pass.key 2048 • Gerar um servidor privado keyopenssl rsa-passin pass : 12345 -in server.pass.key -out server.key • Gerar uma solicitação de assinatura. Serão feitas perguntas adicionais sobre sua empresa. Basta pressionar o botão "Enter" todo o tempo.openssl req -new -key server.key -out server.csr • Gerar o certificateopenssl x509 -req -days 1095 -in server.csr -signkey server.key -out server.crt Agora você tem dois arquivos, o certificado (server.crt) e a chave privada (server.key). Copie-os para a pasta de raiz do servidor de sinalização. Para habilitar a conexão de soquete segura, modifique nosso servidor de sinalização. //require o módulo do sistema de arquivos var fs = require ('fs'); Var httpServ = require ('https'); //https://github.com/visionmedia/superagent/issues/205 process.env.NODE\_TLS\_REJECT\_UNAUTHORIZED = "0"; // o servidor seguro irá ligar à porta 9090 var cfg = {port: 9090, ssl\_key: 'server.key', ssl\_cert: 'server.crt'}; // no caso de solicitação http apenas envie de volta "OK" var processRequest = função (req, res) {res.writeHead (200); Res.end ("OK"); }; // crie o nosso servidor com o var var habilitado por SSL = httpServ.createServer ({chave: fs.readFileSync (cfg.ssl\_key), cert: fs.readFileSync (cfg.ssl\_cert)}, processRequest) .listen (cfg.port); // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criando um servidor de websocket na porta 9090 var wss = new WebSocketServer ({server: app}); // todos conectados aos usuários do servidor var users = {}; // requer a biblioteca redis em Node.jsvar redis = require ("redis"); // criando o objeto redis client var redisClient = redis.createClient (); // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {//...ou outro código. No código acima, exigimos que a biblioteca fs leia chave e certificado privado, crie o objeto cfg com A porta de ligação e os caminhos para a chave e o certificado privados. Em seguida, criamos um servidor HTTPS com as nossas chaves juntamente com o servidor WebSocket na porta 9090. Agora abra https: // localhost: 9090 no Opera. Você deve ver o seguinte - Clique no botão "Continue de qualquer maneira". Você deve ver a mensagem "OK". Para testar nosso servidor de sinalização seguro, modificaremos o aplicativo de bate-papo que criamos no tutorial "WebRTC Text Demo". Só precisamos adicionar um campo de senha. É todo o arquivo index.html - <html> <head> <title> WebRTC Text Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap.min.css" /> </ Head> <style> body {background: #eee; padding: 5% 0;} </ style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row "> < Div class = "col-md-4 col-md-offset-4"> <h2> WebRTC Text Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <input type =" text "id =" passwordInput "class =" form-control form-group "placeholder =" Password "required =" "autofocus =" "> <id do botão = "LoginBtn" class = "btn btn-lg btn-primary btnblock"> Inscreva-se </ button> </ div> </ div> </ div> <div id = "callPage" class = "contêiner da página de chamada" > <Div class = "row"> <div class = "col-md-4 col-md-offset-4 text-center"> <div class = "panel panel-primary"> <div class = "painel-título "> Chat de texto </ div> <div id =" chatarea "class =" painel-corpo texto-esquerdo "> </ div> </ div> </ div> </ div> <div class =" row text- Grupo-grupo central "> <div class =" col-md-12 "> <input id =" callToUsernameInput "type =" text "placeholder =" nome de usuário para ligar "/> <button id =" callBtn "class =" btn -success btn "> Chamar </ button> <botão id =" hangUpBtn "class =" btn-dan Ger btn ">> <div class =" col-md-12 "> <input id =" msgInput "type = "Texto" espaço reservado = "mensagem" /> <botão id = "sendMsgBtn" class = "btn-sucesso btn"> Enviar </ button> </ div> </ div> </ div> <script src = "cliente. Js "> </ script> </ body> </ html> Também precisamos habilitar uma conexão de soquete segura no arquivo client.js através desta linha var conn = new WebSocket ('wss: // localhost: 9090'); . Observe o protocolo wss. Em seguida, o manipulador do botão de login deve ser modificado para enviar a senha juntamente com o nome de usuário -loginBtn.addEventListener ("clique", função (evento) {nome = nome\_de\_usuário.valor; var pwd = passwordInput.value; if (name.length> 0) { Enviar ({tipo: "login", nome: nome, senha: pwd});}}); O seguinte é o arquivo client.js inteiro - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('wss: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var passwordInput = document.querySelector ('# passwordInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInpUt = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Dados varCanal; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; var pwd = passwordInput.value; if (name.length> 0) {send ({type: " Login ", nome: nome, senha: pwd});}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... nome de usuário ou senha incorretos"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; }; // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = nome + ":" + val + "<br />" ; // enviando uma mensagem para um dado de dados de pares conectadosChannel.send (val); msgInput.value = "";}); agora execute nosso servidor de sinalização seguro através do servidor de nó. Execute o nó estático dentro da pasta de demonstração de bate-papo modificada. Abra o localhost: 8080 em duas guias do navegador. Tente iniciar sessão. Lembre-se de que apenas "user1" com "password1" e "user2" com "password2" podem entrar. Em seguida, estabeleça o RTCPeerConnection (chame outro usuário) e tente enviar uma mensagem. O seguinte é o código completo do nosso servidor de sinalização seguro - // requer o módulo do sistema de arquivos var fs = require ('fs'); Var httpServ = require ('https'); // https: //github.com/visionmedia/superagent/issues/205 process.env.NODE\_TLS\_REJECT\_UNAUTHORIZED = "0"; // o servidor seguro irá ligar à porta 9090 var cfg = {port: 9090, ssl\_key: 'server.key', ssl\_cert: 'server.crt'}; // no caso de solicitação http apenas envie de volta "OK" var processRequest = função (req, res) {res.writeHead (200); Res.end ("OK"); }; // crie o nosso servidor com o var var habilitado por SSL = httpServ.createServer ({chave: fs.readFileSync (cfg.ssl\_key), cert: fs.readFileSync (cfg.ssl\_cert)}, processRequest) .listen (cfg.port); // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criando um servidor de websocket na porta 9090 var wss = new WebSocketServer ({server: app}); // todos conectados aos usuários do servidor var users = {}; // requer a biblioteca redis em Node.js var redis = require ("redis"); // criando o objeto do cliente redis var redisClient = redis.createClient (); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // Quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console. Log ("JSON inválido"); data = {};} // verifique seUm usuário é autenticado se (data.type! = "Login") {// se o usuário não for autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", mensagem: "You are not authenticated" }); Retorna; }} // tipo de comutação do parâmetro de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário logado:", data.name); // obter senha para este nome de usuário a partir do banco de dados redis redisClient.get (data.name, function (err, reply) {// verificar se a senha coincide com a armazenada em redis var loginSuccess = reply === data.password; // Se alguém estiver logado com esse nome de usuário ou senha incorreta, recusar se (usuários [data.name] ||! LoginSuccess) {sendTo (connection, {type: "login", success: false}); else else // // Conexão do usuário nos usuários do servidor [data.name] = conexão; connection.name = data.name; connection.isAuth = true; sendTo (conexão, {digite: "login", sucesso: true});}}); pausa; Caso "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); = Usuários [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {apagar usuários [connection.name];}}); Connection.send ("Olá do servidor"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } SumárioNeste capítulo, adicionamos autenticação de usuário ao nosso servidor de sinalização. Também aprendemos a criar certificados SSL auto-assinados e usá-los no escopo de aplicativos WebRTC .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Guia Rápido \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Publicidades \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página AnteriorPágina seguinte \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_WebRTC - Visão geral A Web não é mais um estranho para a comunicação em tempo real como WebRTC (Web Real-Time Communication ) Entra em jogo. Embora tenha sido lançado em maio de 2011, ainda está em desenvolvimento e seus padrões estão mudando. Um conjunto de protocolos é padronizado pela comunicação em tempo real no grupo de trabalho WEB-browsers http://tools.ietf.org/wg/rtcweb/ do IETF (Internet Engineering Task Force) enquanto novos conjuntos de APIs são padronizados pelo Grupo de Trabalho de Comunicações em Tempo Real da Web em http://www.w3.org/2011/04/webrtc/ do W3C (World Wide Web Consortium). Com a aparência do WebRTC, as aplicações web modernas podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas. O Basic SchemeWebRTC permite que você configure conexões peer-to-peer com outros navegadores de maneira rápida e fácil. Para criar tal aplicativo a partir do zero, você precisaria de uma grande quantidade de frameworks e bibliotecas que lidam com problemas típicos, como perda de dados, queda de conexão e NAT traversal. Com o WebRTC, tudo isso vem incorporado no navegador fora da caixa. Esta tecnologia não precisa de plugins ou de software de terceiros. É de código aberto e seu código fonte está disponível gratuitamente em http://www.webrtc.org/. A API WebRTC inclui captura de mídia, codificação e decodificação de áudio e vídeo, camada de transporte e gerenciamento de sessão. Captura de mídia O primeiro passo é obter acesso à câmera e ao microfone do dispositivo do usuário. Detectamos o tipo de dispositivos disponíveis, obtenemos permissão do usuário para acessar esses dispositivos e gerenciamos o stream.Encoding e decodificação de áudio e vídeo. Não é uma tarefa fácil enviar uma transmissão de dados de áudio e vídeo pela Internet. É aí que a codificação e a decodificação são usadas. Este é o processo de dividir quadros de vídeo e ondas de áudio em pedaços menores e comprimireles. Este algoritmo é chamado de codec. Há uma enorme quantidade de codecs diferentes, que são mantidos por diferentes empresas com diferentes objetivos de negócios. Há também muitos codecs dentro da WebRTC como H.264, iSAC, Opus e VP8. Quando dois navegadores se conectam, eles escolhem o melhor codec suportado entre dois usuários. Felizmente, a WebRTC faz a maior parte da codificação nos bastidores. Camada de transporte A camada de transporte administra a ordem dos pacotes, lida com a perda de pacotes e liga-se a outros usuários. Novamente a WebRTC API nos dá um acesso fácil aos eventos que nos dizem quando há problemas com a conexão. Gerenciamento de sessão O gerenciamento de sessão trata de gerenciar, abrir e organizar conexões. Isso é comumente chamado de sinalização. Se você transferir fluxos de áudio e vídeo para o usuário, também faz sentido transferir dados colaterais. Isso é feito pela RTCDataChannel API. Os engenheiros de empresas como Google, Mozilla, Opera e outros fizeram um ótimo trabalho para trazer essa experiência em tempo real para a compatibilidade Web.Browser. Os padrões WebRTC são um dos desenvolvimentos mais rápidos na web, então Isso não significa que todos os navegadores suportam todos os mesmos recursos ao mesmo tempo. Para verificar se o seu navegador suporta WebRTC ou não, você pode visitar http://caniuse.com/#feat=rtcpeerconnection. Ao longo de todos os tutoriais, recomendo que você use o Chrome para todos os exemplos. Preparando o WebRTCLet começa a usar o WebRTC agora. Navegue no seu navegador para o site de demonstração em https://apprtc.appspot.com/ Clique no botão "JOIN". Você deve ver uma notificação suspensa. Clique no botão "Permitir" para começar a transmitir o seu vídeo e áudio para a página da Web. Você deve ver um fluxo de vídeo de si mesmo. Agora abra o URL que você está atualmente em uma nova aba do navegador e clique em "Inscreva-se". Você deve ver dois fluxos de vídeo - um do seu primeiro cliente e outro do segundo. Agora, você deve entender por que o WebRTC é uma ferramenta poderosa. Use Casos A web em tempo real abre a porta para uma nova gama de aplicativos, incluindo bate-papo baseado em texto, compartilhamento de tela e arquivos, jogos, bate-papo de vídeo e muito mais. Além da comunicação, você pode usar o WebRTC para outros fins como: • marketing em tempo real • publicidade em tempo real • comunicações de back office (CRM, ERP, SCM, FFM) • gerenciamento de RH • redes sociais • serviços de namoro • consultas médicas on-line • serviços financeiros • vigilância • jogos multiplayer • transmissão ao vivo • e-learningSummaryNow você deve ter uma compreensão clara do termo WebRTC. Você também deve ter uma idéia de quais tipos de aplicativos podem ser criados com WebRTC, como você já tentou em seu navegador. Para resumir, o WebRTC é uma tecnologia bastante útil.WebRTC - Arquitetura A arquitetura WebRTC geral possui um grande nível de complexidade. Aqui você pode encontrar três camadas diferentes - • API para desenvolvedores web - esta camada contém todos os desenvolvedores web da API necessários, incluindo objetos RTCPeerConnection, RTCDataChannel e MediaStrean. • API para fabricantes de navegadores • API Overridable, quais fabricantes de navegador podem se conectar. Componentes de transporte Permitem estabelecer conexões em vários tipos de redes, enquanto os mecanismos de voz e vídeo são responsáveis ​​por transferir fluxos de áudio e vídeo de uma placa de som e câmera para a rede. Para os desenvolvedores da Web, a parte mais importante é a API da WebRTC. Se analisarmos a arquitetura WebRTC do lado do cliente-servidor, podemos ver que um dos modelos mais usados ​​é inspirado no Trapezoid SIP (Session Initiation Protocol). Neste modelo, ambos os dispositivos estão executando uma aplicação web de diferentes servidores. O objeto RTCPeerConnection configura fluxos para que eles possam se conectar uns aos outros, peer-to-peer. Esta sinalização é feita via HTTP ou WebSockets. Mas o modelo mais comumente usado é o Triângulo. Neste modelo, ambos os dispositivos usam a mesma aplicação web. Dá ao desenvolvedor web mais flexibilidade ao gerenciar conexões de usuário. O WebRTC APIIt consiste em alguns objetos javascript principais - • RTCPeerConnection • MediaStream • RTCDataChannelO objeto RTCPeerConnection Este objeto é o ponto de entrada principal para a API WebRTC. Isso nos ajuda a conectar-se a pares, inicializar conexões e anexar fluxos de mídia. Ele também gerencia uma conexão UDP com outro usuário. A tarefa principal do objeto RTCPeerConnection é configurar e criar uma conexão entre pares. Podemos facilmente ligar chaves pontos da conexão porque este objeto dispara um conjunto de eventos quando eles aparecem. Esses eventos dão acesso à configuração de nossa conexão. O RTCPeerConnection é um objeto simples de javascript, que você pode simplesmente criar dessa maneira - [código] var conn = new RTCPeerConnection (conf); Conn.onaddstream = function (stream) {// use stream here}; [/ Code] O objeto RTCPeerConnection aceita um parâmetro conf, que abordaremos mais adiante nestes tutoriais. O evento onaddstream é disparado quando o usuário remoto adiciona um fluxo de vídeo ou áudio para sua conexão entre pares. Os navegadores APediaMediaStream AP fornecem acesso de desenvolvedor à API getUserMedia, também kComo a API MediaStream. Existem três pontos-chave de funcionalidade: • Oferece um acesso de desenvolvedor a um objeto de fluxo que representa fluxos de vídeo e áudio • Gerencia a seleção de dispositivos de usuário de entrada no caso de um usuário possuir várias câmeras ou microfones no dispositivo • Fornece uma segurança Nível perguntando ao usuário todo o tempo que ele quer buscar s streamPara testar esta API, vamos criar uma página HTML simples. Ele mostrará um único elemento <video>, pergunte à permissão do usuário para usar a câmera e mostrar uma transmissão ao vivo da câmera na página. Crie um arquivo index.html e adicione - [código] <html> <head> <meta charset = "utf-8"> </ head> <body> <video playplay> </ video> <script src = "cliente. Js "> </ script> </ body> </ html> [/ code] Em seguida, adicione um arquivo client.js - [code] // verifica se o navegador suporta a função WebRTC hasUserMedia () {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia || Navigator.msGetUserMedia; Retornar !! navigator.getUserMedia; } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia || Navigator.msGetUserMedia; // obtenha os fluxos de vídeo e áudio do navegador da câmera do usuário.getUserMedia ({video: true, audio: true}, função (stream) {var video = document.querySelector ('video'); // insira o fluxo na tag de vídeo Video.src = window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("Erro. ​​WebRTC não é suportado!"); } [/ Code] Agora, abra o index.html e você deve ver o fluxo de vídeo exibindo seu rosto. Mas tenha cuidado, porque o WebRTC funciona apenas do lado do servidor. Se você simplesmente abrir esta página com o navegador, ela não funcionará. Você precisa hospedar esses arquivos nos servidores Apache ou Node ou qual prefere. O objeto RTCDataChannelApere o envio de fluxos de mídia entre pares, você também pode enviar dados adicionais usando a API DataChannel. Esta API é tão simples quanto a API MediaStream. O principal trabalho é criar um canal proveniente de um objeto RTCPeerConnection existente - [código] var peerConn = new RTCPeerConnection (); // estabelecendo conexão entre pares // ... // fim de estabelecer conexão de pares var dataChannel = peerConnection.createDataChannel ("myChannel", dataChannelOptions); // aqui podemos começar a enviar mensagens diretas para outro peer [/ code] Isto é tudo o que você precisava, apenas duas linhas de código. Tudo o resto é feito na camada interna do navegador. Você pode criar um canal em qualquer conexão entre pares até que o objeto RTCPeerConnection esteja fechado.SummaryVocê deve ter uma compreensão firme da arquitetura WebRTC. Também cobrimos as APIs MediaStream, RTCPeerConnection e RTCDataChannel. A WebRTC API é um alvo em movimento, portanto, mantenha sempre as últimas especificações.WebRTC - Environment Antes de começarmos a desenvolver nossas aplicações WebRTC, devemos definir nosso ambiente de codificação. Em primeiro lugar, você deve ter um editor de texto ou IDE onde você pode editar HTML e Javascript. Há chances de você ter escolhido o preferido porque você está lendo este tutorial. Quanto a mim, estou usando o WebStorm IDE. Você pode baixar sua versão de avaliação em https://www.jetbrains.com/webstorm/. Eu também estou usando o Linux Mint como meu SO de escolha. O outro requisito para aplicativos comuns da WebRTC é ter um servidor para hospedar os arquivos HTML e Javascript. O código não funcionará apenas clicando duas vezes nos arquivos porque o navegador não tem permissão para se conectar a câmeras e microfones, a menos que os arquivos sejam atendidos por um servidor real. Isso é feito, obviamente, devido aos problemas de segurança. Há toneladas de servidores web diferentes, mas neste tutorial, vamos usar o Node.js com o nó estático - • Visite https://nodejs.org/en/ e faça o download A última versão do Node.js. • Descompactá-lo no diretório / usr / local / nodejs. • Abra o arquivo /home/YOUR\_USERNAME/.file de perfil e adicione a seguinte linha ao final - export PATH = $ PATH: / usr / local / Nodejs / bin • Pode reiniciar o computador ou executar a fonte /home/YOUR\_USERNAME/.profile. Agora, o comando do nó deve estar disponível a partir da linha de comando. O comando npm também está disponível. NMP é o gerenciador de pacotes para Node.js. Você pode aprender mais em https://www.npmjs.com/.• Abrir um terminal e executar sudo npm install -g node-static. Isso irá instalar o servidor web estático para Node.js. • Agora navegue até qualquer diretório que contenha os arquivos HTML e execute o comando estático dentro do diretório para iniciar seu servidor da Web. • Você pode navegar para http: // localhost: 8080 para ver Seus arquivos. Há outra maneira de instalar nodejs. Basta executar sudo apt-get install nodejs na janela do terminal. Para testar a instalação do Node.js, abra seu terminal e execute o nodecommand. Digite alguns comandos para verificar como ele funciona - Node.js executa arquivos JavaScript, bem como comandos digitados no terminal. Crie um arquivo index.js com o seguinte conteúdo -console.log ("Testing Node.js"); em seguida, execute o comando do índice do nó. Você verá o seguinte: ao construir nosso servidor de sinalização, usaremos uma biblioteca do WebSockets para Node.js. Para instalarNa execução npm instalar ws no terminal.Para testar o nosso servidor de sinalização, usaremos o utilitário wscat. Para instalá-lo, execute npm instale -g wscat na sua janela de terminal. Não há protocolos e descrição1 Os protocolos WebRTC Os aplicativos do WebBTC usam UDP (User Datagram Protocol) como o protocolo de transporte. A maioria dos aplicativos da Web hoje são construídos com o uso do protocolo TCP Protocol (Protocolo de Controle de Transmissão) 2 Session Description Protocol O SDP é uma parte importante do WebRTC. É um protocolo destinado a descrever as sessões de comunicação de mídia.3 Ao encontrar uma RouteIn para se conectar a outro usuário, você deve encontrar um caminho claro em sua própria rede e na rede do outro usuário. Mas há chances de que a rede que você está usando tenha vários níveis de controle de acesso para evitar problemas de segurança.4 Protocolo de transmissão de controle de fluxo Com a conexão entre pares, temos a capacidade de enviar rapidamente dados de vídeo e áudio. O protocolo SCTP é usado hoje para enviar dados blob em cima da nossa conexão de configuração de pares no momento, usando o objeto RTCDataChannel.SummaryNeste capítulo, nós abordamos várias das tecnologias que permitem conexões pares, como UDP, TCP, STUN, TURN, ICE , E SCTP. Você deve agora ter uma compreensão ao nível da superfície de como o SDP funciona e seus casos de uso. APIWebRTC - MediaStream APIs A MediaStream API foi projetada para facilitar o acesso aos fluxos de mídia de câmeras e microfones locais. O método getUserMedia () é a principal forma de acessar dispositivos de entrada locais. A API possui alguns pontos-chave: • Um fluxo de mídia em tempo real é representado por um objeto de fluxo na forma de vídeo ou áudio • Ele fornece um nível de segurança através de Permissões de usuário perguntando ao usuário antes que um aplicativo da Web possa começar a buscar um fluxo • A seleção de dispositivos de entrada é gerenciada pela API MediaStream (por exemplo, quando há duas câmeras ou microfones conectados ao dispositivo) Cada objeto MediaStream inclui vários objetos MediaStreamTrack. Eles representam vídeo e áudio de diferentes dispositivos de entrada. Cada objeto MediaStreamTrack pode incluir vários canais (canais de áudio direito e esquerdo). Estas são as partes mais pequenas definidas pela MediaStream API. Existem duas maneiras de produzir objetos MediaStream. Primeiro, podemos renderizar a saída para um elemento de vídeo ou áudio. Em segundo lugar, podemos enviar saída para o objeto RTCPeerConnection, que, em seguida, enviá-lo para um ponto remoto. Usando o MediaStream APILet, crie um aplicativo WebRTC simples. Ele mostrará um elemento de vídeo na tela, pergunte ao usuário permissão para usar a câmera e mostra um fluxo de vídeo ao vivo no navegador. Crie um arquivo index.html - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <video playplay> </ video> < Script src = "client.js"> </ script> </ body> </ html> Em seguida, crie o arquivo client.js e adicione o seguinte: function hasUserMedia () {// verifique se o navegador aceita o retorno da WebRTC !! (Navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e de áudio navigator.getUserMedia ({video: true, audio: true}, function (stream) {var video = document.querySelector ('video'); // inserindo nosso fluxo na tag de vídeo video.src = Window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("WebRTC não é suportado"); } Aqui criamos a função hasUserMedia () que verifica se o WebRTC é suportado ou não. Então, acessamos a função getUserMedia onde o segundo parâmetro é um retorno de chamada que aceita o fluxo que vem do dispositivo do usuário. Em seguida, carregamos nosso fluxo no elemento de vídeo usando window.URL.createObjectURL que cria um URL que representa o objeto dado no parâmetro. Agora atualize sua página, clique em Permitir e você deve ver seu rosto na tela. Lembre-se de executar todos os seus scripts usando o servidor web. Nós já instalamos um no Tutorial de ambiente WebRTC. Medição de APIProperties • MediaStream.active (somente leitura) - Retorna verdadeiro se o MediaStream estiver ativo, ou falso de outra forma. • MediaStream.ended (somente leitura, desativado) - Retornar verdadeiro se o final O evento foi disparado no objeto, o que significa que o fluxo foi completamente lido ou falso se o fim do fluxo não foi alcançado. • MediaStream.id (somente leitura) - Um identificador exclusivo para o objeto. • MediaStream.label (Somente leitura, obsoleta) - Um identificador exclusivo atribuído pelo agente do usuário. Você pode ver como as propriedades acima aparecem no meu navegador - Manipuladores de eventos • MediaStream.onactive - Um manipulador para um evento ativo que é acionado quando um objeto MediaStream se torna ativo • MediaStream.onaddtrack - Um manipulador para um evento addtrack que é disparado quando um novo objeto MediaStreamTrack é adicionado. • MediaStream.onended (obsoleto) - Um manipulador para endedevent que é disparado quando a transmissão está terminando. • MediaStream.oninactive - Um manipulador para um evento inativo que é disparado quando um objeto MediaStream se torna inativo. • MediaStReam.onremovetrack - Um manipulador para um evento removetrack que é disparado quando um objeto MediaStreamTrack é removido dele. Métodos • MediaStream.addTrack () - Adiciona o objeto MediaStreamTrack dado como argumento para o MediaStream. Se a faixa já tiver sido adicionada, nada acontece. • MediaStream.clone () - Retorna um clone do objeto MediaStream com uma nova ID. • MediaStream.getAudioTracks () - Retorna uma lista dos objetos MediaStreamTrack de áudio do objeto MediaStream. • MediaStream.getTrackById () - Retorna a faixa por ID. Se o argumento estiver vazio ou o ID não for encontrado, ele retornará nulo. Se você tiver várias faixas com a mesma ID, retorna a primeira. • MediaStream.getTracks () - Retorna uma lista de todos os objetos MediaStreamTrack do objeto MediaStream. • MediaStream.getVideoTracks () - Retorna uma lista dos objetos MediaStreamTrack de video do MediaStream Objeto. • MediaStream.removeTrack () - Remove o MediaStreamTrackobject dado como argumento do MediaStream. Se a faixa já foi removida, nada acontece. Para testar as APIs acima, altere a alteração index.html da seguinte maneira - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf- 8 "/> </ head> <body> <videoplay> </ video> <div> <button id =" btnGetAudioTracks "> getAudioTracks () </ button> </ div> <div> <button id =" btnGetTrackById "> GetTrackById () </ button> </ div> <div> <button id =" btnGetTracks "> getTracks () </ button> </ div> <div> <button id =" btnGetVideoTracks "> getVideoTracks () < / Button> </ div> <div> <button id = "btnRemoveAudioTrack"> removeTrack () - áudio </ button> </ div> <div> <button id = "btnRemoveVideoTrack"> removeTrack () - video </ button > </ Div> <script src = "client.js"> </ script> </ body> </ html> Adicionamos alguns botões para experimentar várias APIs MediaStream. Então, devemos adicionar manipuladores de eventos para o nosso botão recém-criado. Modifique o arquivo client.js dessa maneira - fluxo constante; Função hasUserMedia () {// verifique se o navegador suporta o retorno da WebRTC !! (navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e áudio navigator.getUserMedia ({video: true, audio: true}, função (s) {stream = s; var video = document.querySelector ('video'); // inserindo nosso fluxo no video Etiqueta video.src = window.URL.createObjectURL (stream);}, function (err) {}); } Else {alert ("WebRTC não é suportado"); } BtnGetAudioTracks.addEventListener ("clique", função () {console.log ("getAudioTracks"); console.log (stream.getAudioTracks ());}); BtnGetTrackById.addEventListener ("clique", função () {console.log ("getTrackById"); console.log (stream.getTrackById (stream.getAudioTracks () [0] .id));}); BtnGetTracks.addEventListener ("clique", função () {console.log ("getTracks ()"); console.log (stream.getTracks ());}); BtnGetVideoTracks.addEventListener ("clique", função () {console.log ("getVideoTracks ()"); console.log (stream.getVideoTracks ());}); btnRemoveAudioTrack.addEventListener ("clique", função () {console .log ("removeAudioTrack ()"); stream.removeTrack (stream.getAudioTracks () [0]);}); BtnRemoveVideoTrack.addEventListener ("clique", função () {console.log ("removeVideoTrack ()"); stream.removeTrack (stream.getVideoTracks () [0]);}); agora atualize sua página. Clique no botão getAudioTracks () e, em seguida, clique no botão removeTrack () - audio. A faixa de áudio agora deve ser removida. Em seguida, faça o mesmo para a faixa de vídeo. Se você clicar no botão getTracks (), você deve ver todos os MediaStreamTracks (todas as entradas de vídeo e áudio conectadas). Em seguida, clique no getTrackById () para obter o áudio MediaStreamTrack. Resumo Neste capítulo, criamos um aplicativo WebRTC simples usando a API MediaStream. Agora, você deve ter uma visão geral clara das várias APIs MediaStream que fazem o WebRTC funcionar. APIsWebRTC - RTCPeerConnection API A API RTCPeerConnection é o núcleo da conexão peer-to-peer entre cada um dos navegadores. Para criar os objetos RTCPeerConnection simplesmente writevar pc = RTCPeerConnection (config), onde o argumento config contém pelo menos a chave, iceServers. É uma série de objetos de URL que contém informações sobre os servidores STUN e TURN, usados ​​durante a descoberta dos candidatos do ICE. Você pode encontrar uma lista de servidores STUN públicos disponíveis em code.google.com. Dependendo de se você é o chamador ou o destinatário, o objeto RTCPeerConnection é usado de forma ligeiramente diferente em cada lado da conexão. Aqui é um exemplo do fluxo do usuário - • Registre o manipulador onicecandidate. Ele envia qualquer candidato ICE para o outro parceiro, conforme eles são recebidos. • Registre o manipulador de corrente. Ele controla a exibição do fluxo de vídeo uma vez que é recebido do ponto remoto. • Registre o manipulador de mensagens. Seu servidor de sinalização também deve ter um manipulador para mensagens recebidas do outro ponto. Se a mensagem contiver o objeto RTCSessionDescription, ele deve ser adicionado ao objeto RTCPeerConnection usando o setRemoteDMétodo escription (). Se a mensagem contiver o objeto RTCIceCandidate, ele deve ser adicionado ao objeto RTCPeerConnection usando o método addIceCandidate (). • Use getUserMedia () para configurar seu fluxo de mídia local e adicione-o ao objeto RTCPeerConnection usando o método addStream (). Comece o processo de negociação de oferta / resposta. Este é o único passo em que o fluxo do chamador é diferente do do chamado. O chamador inicia a negociação usando o método createOffer () e registra uma chamada de retorno que recebe o objeto RTCSessionDescription. Em seguida, esse retorno de chamada deve adicionar este objeto RTCSessionDescription ao seu objeto RTCPeerConnection usando setLocalDescription (). E, finalmente, o chamador deve enviar este RTCSessionDescription para o ponto remoto usando o servidor de sinalização. O destinatário, por outro, registra o mesmo retorno de chamada, mas no método createAnswer (). Observe que o fluxo do destinatário é iniciado somente após a oferta ser recebida do chamador.RTCPeerConnection APIProperties • RTCPeerConnection.iceConnectionState (somente leitura) - Retorna um enum RTCIceConnectionState que descreve o estado da conexão. Um evento iceconnectionstatechange é disparado quando esse valor muda. Os valores possíveis - novo - o agente ICE está aguardando candidatos remotos ou coletando verificação de endereços - o agente ICE tem candidatos remotos, mas não encontrou conexão conectada - o agente ICE encontrou uma conexão utilizável, mas ainda está checando Candidato mais remoto para melhor conexão.o completado - o agente ICE encontrou uma conexão utilizável e parou de testar candidatos remotos.o falhou - o agente ICE verificou todos os candidatos remotos, mas não encontrou uma correspondência para pelo menos um componente.o Desconectado - pelo menos um componente não está mais vivo fechado - o agente ICE está fechado. • RTCPeerConnection.iceGatheringState (somente leitura) - Retorna um enumínio RTCIceGatheringState que descreve o estado de coleta ICE para a conexão - novo - o objeto foi apenas Criado.o recolhimento - o agente ICE está no processo de reunir os candidatos, o agente ICE completou a reunião. • RTCPeerConnection.localDescription (somente leitura) - Retu Rns um RTCSessionDescription descrevendo a sessão local. Pode ser nulo se ainda não tiver sido configurado. • RTCPeerConnection.peerIdentity (somente leitura) - Retorna uma avaliação RTCIdentity. Consiste em um idp (nome de domínio) e um nome que representa a identidade do ponto remoto. • RTCPeerConnection.remoteDescription (somente leitura) - Retorna um RTCSessionDescription descrevendo a sessão remota. Pode ser nulo se ainda não tiver sido configurado. • RTCPeerConnection.signalingState (somente leitura) - Retorna um enumínio RTCSignalingState que descreve o estado de sinalização da conexão local. Este estado descreve a oferta SDP. Um evento signalingstatechange é disparado quando esse valor muda. Os valores possíveis -o estáveis ​​- O estado inicial. Não existe nenhuma oferta de SDP / troca de respostas em andamento; tem-oferta local - o lado local da conexão aplicou localmente uma oferta SDP; tem-oferta remota; o lado remoto da conexão aplicou localmente uma oferta SDP .o have-local-pranswer - uma oferta SDP remota foi aplicada e um prensel SDP aplicado localmente tem-remoção remota - um SDP local foi aplicado e um prensor SDP aplicado remotamente.o fechou - a conexão é Closed.Event HandlersGiven abaixo são os manipuladores de eventos comumente usados ​​de RTCPeerConnection.MethodsGiven abaixo são os métodos comumente usados ​​de RTCPeerConnection.Establishing a ConnectionNow vamos criar um exemplo de aplicação. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor de nó". Serão duas entradas de texto na página, uma para um login e outra para um nome de usuário para o qual queremos conectar. Crie um arquivo index.html e adicione o seguinte código - <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <div> <input type = "text "Id =" loginInput "/> <button id =" loginBtn "> Login </ button> </ div> <div> <input type =" text "id =" otherUsernameInput "/> <button id =" connectToOtherUsernameBtn "> Estabeleça a conexão </ button> </ div> <script src = "client2.js"> </ script> </ body> </ html> Você pode ver que adicionamos a entrada de texto para um login, o botão de login , A entrada de texto para o outro nome de usuário do peer e o botão conectar-para-ele. Agora, crie um arquivo client.js e adicione o seguinte código -var connection = new WebSocket ('ws: // localhost: 9090'); Var name = ""; Var loginInput = document.querySelector ('# loginInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var otherUsernameInput = document.querySelector ('# otherUsernameInput'); Var connectToOtherUsernameBtn = document.querySelector ('# connectToOtherUsernameBtn'); Var connectedUser, myConnection; // quando um usuário clica no botão de login loginBtn.addEventListener ("clique", função (evento) {name = loginInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); // lida com mensagens do servidor connection.onmessage = function (messagE) {console.log ("Got message", message.data); Var data = JSON.parse (message.data); Switch (data.type) {case "login": onLogin (data.success); pausa; Caso "oferta": onOffer (data.offer, data.name); pausa; Caso "resposta": onAnswer (data.answer); pausa; Caso "candidato": onCandidate (data.candidate); pausa; Padrão: intervalo; }}; // quando um usuário entra na função onLogin (sucesso) {if (success === false) {alert ("oops ... tente um nome de usuário diferente"); } Else {// criando nosso objeto RTCPeerConnection var configuration = {"iceServers": [{"url": "stun: stun.1.google.com: 19302"}]}; MyConnection = new webkitRTCPeerConnection (configuração); Console.log ("o objeto RTCPeerConnection foi criado"); Console.log (myConnection); // configuração de manipulação de gelo // quando o navegador encontra um candidato de gelo, enviamos para outro ponto myConnection.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate }); }}; }}; Connection.onopen = function () {console.log ("Conectado"); }; Connection.onerror = function (err) {console.log ("Got error", err); }; // Alias ​​para enviar mensagens no formato JSON enviar (mensagem) {if (connectedUser) {message.name = connectedUser; } Connection.send (JSON.stringify (mensagem)); }; Você pode ver que estabelecemos uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login, o aplicativo envia seu nome de usuário para o servidor. Se o login for bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o manipulador onicecandidate da configuração que envia todos os icecandidatos encontrados para o outro ponto. Agora abra a página e tente fazer o login. Você deve ver a seguinte saída do console - O próximo passo é criar uma oferta para o outro par. Adicione o seguinte código ao seu arquivo client.js - // configure uma conexão peer com outro usuário connectToOtherUsernameBtn.addEventListener ("clique", function () {var otherUsername = otherUsernameInput.value; connectedUser = otherUsername; if (otherUsername.length> 0 ) {// faça uma oferta myConnection.createOffer (função (oferta) {console.log (); enviar ({tipo: "oferta", oferta: oferta}); myConnection.setLocalDescription (oferta);}, função (erro) {Alert ("Ocorreu um erro.");});}}); // quando alguém quer nos chamar de função onOffer (oferta, nome) {connectedUser = nome; MyConnection.setRemoteDescription (novo RTCSessionDescription (oferta)); MyConnection.createAnswer (função (resposta) {myConnection.setLocalDescription (resposta); send ({type: "answer", answer: answer});}, function (error) {alert ("oops ... error");} ); } // quando outro usuário responde a nossa função de oferta onAnswer (answer) {myConnection.setRemoteDescription (novo RTCSessionDescription (resposta)); // Quando obtivemos um candidato de gelo de outra função de usuário onCandidate (candidato) {myConnection.addIceCandidate (novo RTCIceCandidate (candidato)); } Você pode ver que, quando um usuário clica no botão "Estabelecer conexão", o aplicativo faz uma oferta SDP para o outro par. Nós também definimos onAnswer e onCandidate manipuladores. Recarregue a sua página, abra-a em duas guias, faça o login com dois usuários e tente estabelecer uma conexão entre eles. Você deve ver a seguinte saída do console - Agora a conexão peer-to-peer é estabelecida. Nos próximos tutoriais, adicionaremos fluxos de vídeo e áudio, bem como suporte de bate-papo de texto. O WebRTC - RTCDataChannel APIsWebRTC não é apenas bom na transferência de transmissões de áudio e vídeo, mas em qualquer dado arbitrário que possamos ter. Este é o lugar onde o objeto RTCDataChannel entra em jogo.RTCDataChannel APIProperties • RTCDataChannel.label (somente leitura) - Retorna uma string contendo o nome do canal de dados. • RTCDataChannel.ordered (somente leitura) - Retorna verdadeiro se a ordem de entrega das mensagens for Garantido ou falso se não for garantido. • RTCDataChannel.protocol (somente leitura) - Retorna uma string contendo o nome do subprotocolo usado para este canal. • RTCDataChannel.id (somente leitura) - Retorna um ID exclusivo para o canal que está configurado no Criação do objeto RTCDataChannel. • RTCDataChannel.readyState (somente leitura) - Retorna o enumínio RTCDataChannelState que representa o estado da conexão. Os possíveis valores -o conexão - Indica que a conexão ainda não está ativa. Este é o estado inicial.o aberto - Indica que a conexão está sendo executada.o fechamento - Indica que a conexão está em processo de desligamento. As mensagens em cache estão em processo de envio ou recebimento, mas nenhuma tarefa recém-criada está aceitando.o fechado - Indica que a conexão não pôde ser estabelecida ou foi encerrada. • RTCDataChannel.bufferedAmount (somente leitura) - Retorna o valor De bytes que foram encaminhados para envio. Esta é a quantidade de dados que ainda não foram enviados através de RTCDataChannel.send (). • RTCDataChannel.bufferedAmountLowThreshold - Retorna o número de bytes em que o RTCDataChannel.bufferedAmount é ocupado como baixo. Quando o RTCDataChannel.bufferedAmount diminui abaixo deste limite, o evento bufferedamountlow é disparado. • RTCDataChannel.binaryType - Retorna oE tipo dos dados binários transmitidos pela conexão. Pode ser "blob" ou "arraybuffer". • RTCDataChannel.maxPacketLifeType (somente leitura) - Retorna um curto não assinado que indica o comprimento em milissegundos da janela quando a mensagem está no modo não confiável. • RTCDataChannel.maxRetransmits (somente leitura) - Retorna um curto não assinado que indica o número máximo de vezes que um canal irá retransmitir dados se não for entregue. • RTCDataChannel.negotiated (somente leitura) - Retorna um booleano que indica se o canal foi negociado pelo agente do usuário ou por O aplicativo. • RTCDataChannel.reliable (somente leitura) - Retorna um booleano que indica que a conexão pode enviar mensagens em modo não confiável. • RTCDataChannel.stream (somente leitura) - Sinônimo de RTCDataChannel.idEvent Handlers • RTCDataChannel.onopen - Esse manipulador de eventos É chamado quando o evento aberto é disparado. Este evento é enviado quando a conexão de dados foi estabelecida. • RTCDataChannel.onmessage - Este manipulador de eventos é chamado quando o evento da mensagem é disparado. O evento é enviado quando uma mensagem está disponível no canal de dados. • RTCDataChannel.onbufferedamountlow - Este manipulador de eventos é chamado quando o evento bufferedamoutlow é disparado. Este evento é enviado quando RTCDataChannel.bufferedAmount diminui abaixo da propriedade RTCDataChannel.bufferedAmountLowThreshold. • RTCDataChannel.onclose - Este manipulador de eventos é chamado quando o evento de fechamento é disparado. Este evento é enviado quando a conexão de dados foi fechada. • RTCDataChannel.onerror - Este manipulador de eventos é chamado quando o evento de erro é disparado. Este evento é enviado quando um erro foi encontrado. Métodos • RTCDataChannel.close () - Fecha o canal de dados. • RTCDataChannel.send () - Envia os dados no parâmetro através do canal. Os dados podem ser um blob, uma string, um ArrayBuffer ou um ArrayBufferView.WebRTC - Enviando Mensagens Agora, vamos criar um exemplo simples. Em primeiro lugar, execute o servidor de sinalização que criamos no tutorial "servidor de sinalização" via "servidor de nó". Terão três entradas de texto na página, uma para um login, uma para um nome de usuário e outra para a mensagem que queremos enviar Para o outro par. Crie um arquivo index.html e adicione o seguinte código - <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <div> <input type = "text "Id =" loginInput "/> <button id =" loginBtn "> Login </ button> </ div> <div> <input type =" text "id =" otherUsernameInput "/> <button id =" connectToOtherUsernameBtn "> Estabeleça a conexão </ button> </ div> <div> <input type = "text" id = "msgInput" /> <button id = "sendMsgBtn"> Enviar mensagem de texto </ button> </ div> <script src = "Client.js"> </ script> </ body> </ html> Também adicionamos três botões para login, estabelecendo uma conexão e enviando uma mensagem. Agora, crie um arquivo client.js e adicione o seguinte código -var connection = new WebSocket ('ws: // localhost: 9090'); Var name = ""; var loginInput = document.querySelector ('# loginInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var otherUsernameInput = document.querySelector ('# otherUsernameInput'); Var connectToOtherUsernameBtn = document.querySelector ('# connectToOtherUsernameBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var connectedUser, myConnection, dataChannel; // quando um usuário clica no botão de login loginBtn.addEventListener ("clique", função (evento) {name = loginInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); // lida com mensagens do servidor connection.onmessage = function (message) {console.log ("Got message", message.data); Var data = JSON.parse (message.data); Switch (data.type) {case "login": onLogin (data.success); pausa; Caso "oferta": onOffer (data.offer, data.name); pausa; Caso "resposta": onAnswer (data.answer); pausa; Caso "candidato": onCandidate (data.candidate); pausa; Padrão: intervalo; }}; // quando um usuário entra na função onLogin (sucesso) {if (success === false) {alert ("oops ... tente um nome de usuário diferente"); } Else {// criando nosso objeto RTCPeerConnection var configuration = {"iceServers": [{"url": "stun: stun.1.google.com: 19302"}]}; MyConnection = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); Console.log ("o objeto RTCPeerConnection foi criado"); Console.log (myConnection); // configuração de manipulação de gelo // quando o navegador encontra um candidato de gelo, enviamos para outro ponto myConnection.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate }); }}; OpenDataChannel (); }}; Connection.onopen = function () {console.log ("Conectado"); }; Connection.onerror = function (err) {console.log ("Got error", err); }; // Alias ​​para enviar mensagens no formato JSON enviar (mensagem) {if (connectedUser) {message.name = connectedUser; } Connection.send (JSON.stringify (mensagem)); }; Você pode ver que estabelecemos uma conexão de soquete para o nosso servidor de sinalização. Quando um usuário clica no botão de login, o aplicativo envia seu usernaEu para o servidor. Se o login for bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o manipulador onicecandidate da configuração que envia todos os icecandidatos encontrados para o outro ponto. Ele também executa a função openDataChannel () que cria um dataChannel. Observe que ao criar o objeto RTCPeerConnection o segundo argumento no construtor opcional: [{RtpDataChannels: true}] é obrigatório se você estiver usando o Chrome ou o Opera. O próximo passo é criar uma oferta para o outro ponto. Adicione o seguinte código ao seu arquivo client.js - // configure uma conexão peer com outro usuário connectToOtherUsernameBtn.addEventListener ("clique", function () {var otherUsername = otherUsernameInput.value; connectedUser = otherUsername; if (otherUsername.length> 0 ) {// faça uma oferta myConnection.createOffer (função (oferta) {console.log (); enviar ({tipo: "oferta", oferta: oferta}); myConnection.setLocalDescription (oferta);}, função (erro) {Alert ("Ocorreu um erro.");});}}); // quando alguém quer nos chamar de função onOffer (oferta, nome) {connectedUser = nome; MyConnection.setRemoteDescription (novo RTCSessionDescription (oferta)); MyConnection.createAnswer (função (resposta) {myConnection.setLocalDescription (resposta); send ({type: "answer", answer: answer});}, function (error) {alert ("oops ... error");} ); } // quando outro usuário responde a nossa função de oferta onAnswer (answer) {myConnection.setRemoteDescription (novo RTCSessionDescription (resposta)); // Quando obtivemos um candidato de gelo de outra função de usuário onCandidate (candidato) {myConnection.addIceCandidate (novo RTCIceCandidate (candidato)); } Você pode ver que, quando um usuário clica no botão "Estabelecer conexão", o aplicativo faz uma oferta SDP para o outro par. Nós também definimos onAnswer e onCandidate manipuladores. Finalmente, vamos implementar a função openDataChannel () que cria o nosso dataChannel. Adicione o seguinte código ao seu arquivo client.js - // criando função de canal de dados openDataChannel () {var dataChannelOptions = {reliable: true}; DataChannel = myConnection.createDataChannel ("myDataChannel", dataChannelOptions); DataChannel.onerror = function (error) {console.log ("Erro:", erro); }; DataChannel.onmessage = function (event) {console.log ("Got message:", event.data); }; // quando um usuário clicar no botão enviar mensagem sendMsgBtn.addEventListener ("clique", função (evento) {console.log ("enviar mensagem"); var val = msgInput.value; dataChannel.send (val);}) ; Aqui criamos o dataChannel para nossa conexão e adicionamos o manipulador de eventos para o botão "Enviar mensagem". Agora, abra esta página em duas guias, faça o login com dois usuários, estabeleça uma conexão e tente enviar mensagens. Você deve vê-los na saída do console. Observe que o exemplo acima é testado no Opera. Agora você pode ver que o RTCDataChannel é parte extremamente poderosa da API WebRTC. Há muitos outros casos de uso para este objeto, como jogos peer-to-peer ou compartilhamento de arquivos baseado em torrent. O WebRTC - SignalingMost Os aplicativos WebRTC não são apenas capazes de se comunicar através de vídeo e áudio. Eles precisam de muitos outros recursos. Neste capítulo, vamos construir um servidor de sinalização básica.Signação e Negociação Para conectar-se a outro usuário, você deve saber onde ele está localizado na Web. O endereço IP do seu dispositivo permite que os dispositivos habilitados para Internet enviem dados diretamente entre si. O objeto RTCPeerConnection é responsável por isso. Assim que os dispositivos sabem como se encontrar através da Internet, eles começam a trocar dados sobre quais protocolos e codecs cada dispositivo suporta. Para se comunicar com outro usuário, você simplesmente precisa trocar informações de contato e o resto será feito pela WebRTC. O processo de conexão com o outro usuário também é conhecido como sinalização e negociação. Consiste em algumas etapas: • Crie uma lista de potenciais candidatos para uma conexão entre pares. • O usuário ou um aplicativo seleciona um usuário para estabelecer uma conexão. • A camada de sinalização notifica outro usuário que alguém deseja se conectar a ele. Ele pode aceitar ou recusar. • O primeiro usuário é notificado da aceitação da oferta. • O primeiro usuário inicia o RTCPeerConnection com outro usuário. • Ambos os usuários trocam informações de software e hardware através do servidor de sinalização. • Ambos os usuários trocam informações de localização. A conexão é bem-sucedida ou falha. A especificação WebRTC não contém nenhum padrão sobre a troca de informações. Portanto, tenha em mente que o acima é apenas um exemplo de como a sinalização pode acontecer. Você pode usar qualquer protocolo ou tecnologia que você gosta. Construindo o Servidor O servidor que vamos construir pode conectar dois usuários que não estão localizados no mesmo computador. Criaremos o nosso próprio mecanismo de sinalização. Nosso servidor de sinalização permitirá que um usuário ligue para outro. Uma vez que um usuário tenha chamado outro, o servidor passa a oferta, responde, os candidatos do ICE entre eles e configura uma conexão WebRTC. O diagrama acima é o fluxo de mensagens entre usuários ao usar o servidor de sinalização. Em primeiro lugar, cada usuário registra o conhecimentoH o servidor. No nosso caso, este será um nome de usuário de string simples. Uma vez que os usuários se registraram, eles podem ligar-se. O usuário 1 faz uma oferta com o identificador de usuário que deseja chamar. O outro usuário deve responder. Finalmente, os candidatos do ICE são enviados entre os usuários até que eles possam fazer uma conexão. Para criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de pares WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já instalamos. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta ao usuário conectado dizendo "Olá do servidor". Agora, o servidor de nó de execução e o servidor devem começar a ouvir conexões de soquete. Para testar nosso servidor, usaremos o utilitário wscat que também já instalamos. Esta ferramenta ajuda a se conectar diretamente ao servidor WebSocket e a testar os comandos. Execute nosso servidor em uma janela de terminal, abra outro e execute o comando wscat -c ws: // localhost: 9090. Você deve ver o seguinte no lado do cliente - O servidor também deve registrar o usuário conectado - Registro do usuárioNo nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão, então nós sabemos onde enviar mensagens. Vamos mudar o nosso manual de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // tudo conectado ao servidor usersvar users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário enviar uma mensagem com o tipo de login, nós - • Verifique se alguém já fez logon com este nome de usuário • Se assim for , Então, diga ao usuário que ele não iniciou sessão com êxito • Se ninguém estiver usando esse nome de usuário, nós nos adicionamos Ername como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } A função acima garante que todas as nossas mensagens são enviadas no formato JSON. Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); agora vamos testar nosso servidor com o comando de login. Tenha em mente que todas as mensagens devem ser codificadas no formato JSON. Execute nosso servidor e tente fazer o login. Você deve ver algo como isto - Fazendo uma ligação. Após o login bem-sucedido, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// definindo que UserAConectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Respondendo A resposta à resposta tem um padrão similar que usamos no negociador de ofertas. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após a oferta hander -case "answer": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; Você pode ver como isso é semelhante ao manipulador da oferta. Observe que este código segue as funções createOffer e createAnswer no objeto RTCPeerConnection. Agora, podemos testar nosso mecanismo de oferta / resposta. Conecte dois clientes ao mesmo tempo e tente oferecer e responder. Você deve ver o seguinte: neste exemplo, oferecer e responder são strings simples, mas em um aplicativo real, eles serão preenchidos com os dados do SDP. Candidatos à Candidatos. A parte final está lidando com o candidato do ICE entre os usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Ele deve funcionar de forma semelhante aos manipuladores de oferta e resposta. Como ativar a conexão Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .logname, {type: "leave"});}}}}}); agora, se a A conexão termina, nossos usuários serão desconectados. O fechamento será disparado quando um usuário fechar a janela do navegador enquanto ainda estamos em oferta, resposta ou estado candidato. O Servidor de Sinalização CompletoHere é o código inteiro do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws' ).Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; if (conn! = Nul L) {// configuração que UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conN! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Então o trabalho está pronto eo nosso servidor de sinalização está pronto. Lembre-se de que fazer coisas fora de ordem ao fazer uma conexão WebRTC pode causar problemas.SummaryNeste capítulo, nós construímos um servidor de sinalização simples e direto. Atravessamos o processo de sinalização, registro de usuário e mecanismo de oferta / resposta. Também implementamos o envio de candidatos entre usuários.WebRTC - Suporte de navegador A Web está se movendo tão rápido e está sempre melhorando. Novos padrões são criados todos os dias. Os navegadores permitem que as atualizações sejam instaladas sem o usuário saber, então você deve manter-se atualizado com o que está acontecendo no mundo da Web e da WebRTC. Aqui está uma visão geral sobre o que isso faz hoje. Suporte do navegador. Todo o navegador não possui todos os mesmos recursos do WebRTC ao mesmo tempo. Navegadores diferentes podem estar à frente da curva, o que faz com que alguns recursos WebRTC funcionem em um navegador e não outro. O suporte atual para WebRTC no navegador é mostrado na seguinte imagem. Você pode verificar um status de suporte atualizado do WebRTC em http://caniuse.com/#feat=rtcpeerconnection.Chrome, Firefox e Opera. As últimas versões do Chrome, Firefox e Opera nos sistemas operacionais convencionais do PC, como Mac OS X, Windows e Linux, todos apoiam a WebRTC out-of-the-box. E, o mais importante, os engenheiros das equipes de desenvolvedores do Chrome e do Firefox estão trabalhando juntos para corrigir problemas para que esses dois navegadores possam se comunicar facilmente. Os sistemas operacionais Android OSOn Android, os aplicativos WebRTC para o Chrome e o Firefox devem funcionar de forma rápida. Eles são capazes de trabalhar com outros navegadores após a versão Android Ice Cream Sandwich (4.0). Isto é devido ao compartilhamento de código entre versões de desktop e móveis. O AppleApple ainda não fez nenhum anúncio sobre seus planos para suportar o WebRTC no Safari no OS X. Uma das soluções possíveis para aplicações iOS nativas híbridas os para incorporar o código WebRTC diretamente O aplicativo e carregar este aplicativo em um WebView.Internet ExplorerMicrosoft não suporta WebRTC em desktops. Mas eles confirmaram oficialmente que vão implementar ORTC (Object Realtime Communications) em futuras versões do IE (Edge). Eles não estão planejando suportar o WebRTC 1.0. Eles rotularam sua ORTC como WebRTC 1.1, embora seja apenas um aprimoramento da comunidade e não o padrão oficial. Recentemente, eles adicionaram o suporte ORTC à versão mais recente do Microsoft Edge. Você pode aprender mais em https://blogs.windows.com/msedgedev/2015/09/18/ortc-api-is-now-available-in-microsoftedge/.SummaryNotice que o WebRTC é uma coleção de APIs e protocolos, não Uma única API. O suporte para cada um deles está se desenvolvendo em diferentes navegadores e sistemas operacionais em um nível diferente. Uma ótima maneira de verificar o nível mais recente de suporte é através do http://canisue.com. Ele acompanha a adoção de APIs modernas em vários navegadores. Você também pode encontrar as informações mais recentes sobre os suportes do navegador, bem como demonstrações WebRTC em http://www.webrtc.org, que é suportado pelo Mozilla, Google e Opera.WebRTC - Suporte móvel No mundo móvel, o suporte WebRTC não é No mesmo nível que está nos desktops. Os dispositivos móveis têm sua própria maneira, então a WebRTC também é algo diferente nas plataformas móveis. Ao desenvolver um aplicativo WebRTC para área de trabalho, consideramos o uso do Chrome, Firefox ou Opera. Todos eles suportam o WebRTC fora da caixa. Em geral, você só precisa de um navegador e não se preocupe com o hardware da área de trabalho. No mundo móvel, existem três modos possíveis para o WebRTC hoje - • O aplicativo nativo • O aplicativo do navegador • O navegador nativoAndroidIn 2013, o navegador Firefox para Android foi Apresentou o suporte da WebRTC fora da caixa. Agora você pode fazer chamadas de vídeo em dispositivos Android usando o navegador móvel do Firefox. Ele possui três componentes principais do WebRTC - • PeerConnection - permite chamadas entre navegadores • getUserMedia - fornece acesso à câmera e ao microfone • DataChannels - fornece transferência de dados peer-to-peerGoogle O Chrome para Android também oferece suporte para WebRTC. Como você já notou, as características mais interessantes geralmente aparecem no Chrome. No ano passado, o navegador móvel da Opera apareceu com o suporte da WebRTC. Então, para o Android, você tem Chrome, Firefox e Opera. Outros navegadores não suportam WebRTC.iOS Infelizmente, a WebO RTC não é suportado no iOS agora. Embora o WebRTC funcione bem no Mac ao usar o Firefox, o Opera ou o Chrome, ele não é suportado no iOS. Hoje em dia, seu aplicativo WebRTC não funcionará nos dispositivos móveis da Apple fora da caixa. Mas há um navegador - Bowser. É um navegador web desenvolvido pela Ericsson e ele suporta o WebRTC fora da caixa. Você pode verificar sua página inicial em http://www.openwebrtc.org/bowser/.Today, é a única maneira amigável de suportar seu aplicativo WebRTC no iOS. Outra maneira é desenvolver um aplicativo nativo sozinho. Windows PhonesMicrosoft não suporta WebRTC em plataformas móveis. Mas eles confirmaram oficialmente que vão implementar ORTC (Object Realtime Communications) em futuras versões do IE. Eles não estão planejando suportar o WebRTC 1.0. Eles rotularam o seu ORTC como WebRTC 1.1, embora seja apenas um aprimoramento da comunidade e não o padrão oficial. Por isso, os usuários do Windows Phone agora não podem usar aplicativos WebRTC e não há como vencer esta situação. Os aplicativos do BlackberryWebRTC também não são suportados no Blackberry , De qualquer maneira. Utilizando um Navegador Nativo WebRTC O caso mais conveniente e confortável para os usuários utilizar o WebRTC é usar o navegador nativo do dispositivo. Nesse caso, o dispositivo está pronto para trabalhar com configurações adicionais. Hoje em dia, apenas os dispositivos Android que são a versão 4 ou superior fornecem esse recurso. A Apple ainda não mostra nenhuma atividade com o suporte da WebRTC. Portanto, os usuários do Safari não podem usar aplicativos WebRTC. A Microsoft também não apresentou no Windows Phone 8. Usando a WebRTC por meio de aplicativos do navegador. Isso significa usar aplicativos de terceiros (navegadores da Web não-nativos) para fornecer os recursos do WebRTC. Por enquanto, existem dois desses aplicativos de terceiros. Bowser, que é a única maneira de trazer os recursos do WebRTC para o dispositivo iOS e o Opera, que é uma ótima alternativa para a plataforma Android. O resto dos navegadores móveis disponíveis não suporta WebRTC. Aplicativos móveis móveis Como você pode ver, a WebRTC ainda não possui um grande suporte no mundo móvel. Assim, uma das soluções possíveis é desenvolver aplicações nativas que utilizem a API WebRTC. Mas não é a melhor escolha porque o principal recurso WebRTC é uma solução multi-plataforma. De qualquer forma, em alguns casos, esta é a única maneira porque um aplicativo nativo pode utilizar funções ou recursos específicos do dispositivo que não são suportados pelos navegadores HTML5. Conexão de fluxo de vídeo para dispositivos móveis e de mesa O primeiro parâmetro da API getUserMedia espera um objeto de chaves e Valores indicando ao navegador como processar fluxos. Você pode verificar o conjunto completo de restrições em https://tools.ietf.org/html/draft-alvestrand-constraints-resolution-03. Você pode configurar a ração de aspecto de vídeo, frameRate e outros parâmetros opcionais. Suportar dispositivos móveis é uma das maiores dores, porque os dispositivos móveis possuem espaço de tela limitado e recursos limitados. Você pode querer que o dispositivo móvel capture apenas uma resolução de resolução de 2 x 360 ou fluxo de vídeo menor para economizar energia e largura de banda. Usar a string do agente do usuário no navegador é uma boa maneira de testar se o usuário está em um dispositivo móvel ou não. Vamos ver um exemplo. Crie o arquivo index.html - <! DOCTYPE html> <html lang = "en"> <head> <meta charset = "utf-8" /> </ head> <body> <videoplay video> </ video> < Script src = "client.js"> </ script> </ body> </ html> Em seguida, crie o seguinte arquivo client.js - // restrições para o navegador de desktop var desktopConstraints = {video: {mandatory: {maxWidth: 800, MaxHight: 600}}, áudio: true}; // restrições para o navegador móvel var mobileConstraints = {video: {obrigatório: {maxWidth: 480, maxHeight: 320,}}, audio: true} // se um usuário estiver usando um navegador móvel se (/ Android | iPhone | iPad / I.test (navigator.userAgent)) {var constraints = mobileConstraints; } Else {var constraints = desktopConstraints; } Função hasUserMedia () {// verifique se o navegador suporta o retorno da WebRTC! (Navigator.getUserMedia || navigator.webkitGetUserMedia || navigator.mozGetUserMedia); } If (hasUserMedia ()) {navigator.getUserMedia = navigator.getUserMedia || Navigator.webkitGetUserMedia || Navigator.mozGetUserMedia; // habilitando canais de vídeo e de áudio navigator.getUserMedia (restrições, função (fluxo) {var video = document.querySelector ('video'); // inserindo nosso fluxo na tag de vídeo video.src = window.URL.createObjectURL (fluxo );}, Function (err) {}); } Else {alert ("WebRTC não é suportado"); } Execute o servidor web usando o comando estático e abra a página. Você deve ver que é 800x600. Em seguida, abra esta página em uma viewport móvel usando ferramentas cromadas e verifique a resolução. Deve ser de 20%. As restrições são a maneira mais fácil de aumentar o desempenho do seu aplicativo WebRTC.SummaryNeste capítulo, aprendemos sobre os problemas que podem ocorrer ao desenvolver aplicativos WebRTC para dispositivos móveis. Descobrimos diferentes limitações de suporte à API WebRTC em plataformas móveis. Também lançamos um aplicativo de demonstração onde estabelecemos diferentes restrições para desktop eNavegadores móveis.WebRTC - Demo de vídeo Neste capítulo, vamos criar um aplicativo cliente que permita que dois usuários em dispositivos separados se comuniquem usando o WebRTC. Nossa aplicação terá duas páginas. Um para o login e outro para chamar outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples. Servidor de Sinalização Para criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso manipulador de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); Se o usuário enviar uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário • Em caso afirmativo, diga ao usuário que ele não tenha logado com sucesso • Se ninguém estiver usando esse nome de usuário, nós Adicione o nome de usuário como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); Após o login com êxito, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {Connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é o código completo do nosso servidor de sinalização - // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; if (conn! = Null) {// configuração que UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Aplicação de cliente. Uma maneira de testar esta aplicação está abrindo duas guias do navegador e tentando ligar uma. Por fim, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da web. Você pode aprender mais em http://getbootstrap.com/. Crie uma pasta chamada, por exemplo, "videochat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um arquivo package.json (é necessário gerenciar dependências npm) e adicione o seguinte - {"name": "webrtc-videochat", "versão": "0.1.0", "description": "webrtc -videochat "," autor ":" Autor "," licença ":" BSD-2-Clause "} Em seguida, execute o npm install bootstrap. Isso irá instalar oBiblioteca bootstrap na pasta videochat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Video Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } Video {background: black; Borda: 1px sólido cinza; } .call-page {position: relative; Exibição: bloco; Margem: 0 auto; Largura: 500px; Altura: 500px; } #localVideo {width: 150px; Altura: 150px; Posição: absoluta; Topo: 15px; Direito: 15px; } #remoteVideo {width: 500px; Altura: 500px; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Video Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" c lass = "form-control formgroup" placeholder = "Login" requerido = "" autofocus = ""> <botão id = "loginBtn" class = "btn btn-lg btn-primary btnblock"> Inscreva-se </ button> </ div> </ div> </ div> <Div id = "callPage" class = "call-page"> <video id = "localVideo" reprodução automática> </ video> <video id = "remoteVideo" reprodução automática> </ video> <div class = "row text-center "<Div class =" col-md-12 "> <input id =" callToUsernameInput "type =" text "placeholder =" nome de usuário para ligar "/> <button id =" callBtn "class =" btn-successful btn " > Ligar </ button> <button id = "hangUpBtn" class = "btn-danger btn"> Desligar </ button> </ div> </ div> </ div> <script src = "client.js"> </ Script> </ body> </ html> Esta página deve ser familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Você deve ver os dois elementos de vídeo para fluxos de vídeo locais e remotos. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); // esconde a página de chamada callPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {// exibe a página de chamada se o login for bem-sucedido loginPage.style.display = "none"; CallPage.style.display = "block"; // start peer connection}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem-sucedido, mostramos a página de chamada e começamos a configurar uma conexão entre pares. Para iniciar uma conexão entre pares, precisamos - • Obtenha um fluxo da câmera web. • Crie o objeto RTCPeerConnection. Adicione o seguinte código ao " Seletores UI bloqueiam "-var localVideo = document.querySelector ('# localVideo'); Var remoteVideo =Document.querySelector ('# remoteVideo'); Var yourConn; Var stream; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de vídeo local navigator.webkitGetUserMedia ({video: true, audio: true}, function (myStream) {stream = myStream; // exibindo fluxo de vídeo local na página localVideo.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteVideo.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate});}};}}, função ( Erro) {console.log (erro);}); }}; Agora, se você executar o código, a página deve permitir que você faça login e exiba seu fluxo de vídeo local na página. Agora estamos prontos para iniciar uma chamada. Em primeiro lugar, enviamos uma oferta para outro usuário. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando uma chamada callBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // Crie uma oferta yourConn.createOffer (função (oferta) {enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta" );});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remota, lida com respostaAnswer (resposta) {yourConn.setRemoteDescription (novo RTCSessionDescription (answer));}; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }: Adicionamos um manipulador de cliques ao botão Ligar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador onmessage. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O último passo é implementar o recurso de desligamento. Isso deixará de transmitir dados e informará ao outro usuário para fechar a chamada. Adicione o seguinte código - // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteVideo.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null; } Quando o usuário clicar no botão Desligar - • Ele enviará uma mensagem "deixar" para o outro usuário • Ele fechará o RTCPeerConnection e destruirá a conexão localNada execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. Você pode então chamar a guia e desligar a chamada. O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se a nossa sinalização servervar conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = documEnt.querySelector ('# hangUpBtn'); Var localVideo = document.querySelector ('# localVideo'); Var remoteVideo = document.querySelector ('# remoteVideo'); Var yourConn; Fluxo de var; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({ Digite: "login", nome: nome});}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de vídeo local navigator.webkitGetUserMedia ({video: true, audio: true}, function (myStream) {stream = myStream; // exibindo fluxo de vídeo local na página localVideo.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteVideo.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate});}};}}, função ( Erro) {console.log (erro);}); }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função remota do usuário, lidar com resposta (resposta) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteVideo.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null; } Resumo Esta demonstração fornece uma linha de base de recursos que cada aplicativo WebRTC precisa. Para melhorar esta demo, você pode adicionar a identificação do usuário através de plataformas como o Facebook ou o Google, lidar com a entrada do usuário para dados inválidos. Além disso, a conexão WebRTC pode falhar devido a várias razões, como não suportar a tecnologia ou não poder atravessar firewalls. Um valor de trabalho passou a tornar o aplicativo WebRTC estável.WebRTC - Demo de vozNeste capítulo, vamos construir um aplicativo cliente que permita que dois usuários em dispositivos separados se comuniquem usando fluxos de áudio WebRTC. Nossa aplicação terá duas páginas. Um para login e outro para fazer uma ligação de áudio para outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples.Signaling ServerPara criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso controlador de conexão um pouco -Connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); dados = {};}}); Assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário envia uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário. • Se Então, diga ao usuário que ele não iniciou sessão com sucesso. • Se ninguém estiver usando esse nome de usuário, adicionamos usernam E como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão-connection.on ("fechar", função () {if (connection.name) {excluir usuários [connection.name];}}); Após o login bem-sucedido, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } Break; Em primeiro lugar, obtemos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } Break; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão de pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é O código completo do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // tudo ligado aOs usuários do servidor var users = {}; // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de um Conexão de usuário conectada.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido") ; Data = {};} // tipo de comutação da chave de mensagem do usuário (type.type) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário logado", data.name); / / Se alguém estiver logado com este nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: false}); else else {// salvar conexão do usuário nos usuários do servidor [Data.name] = connection; connection.name = data.name; sendTo (conexão, {digite: "login", success: true});} break; case "offer": // para ex. UserA quer ligar UserB console.log ("Enviando oferta para:", data.name); // se UserB existir, envie-lhe detalhes da oferta var conn = users [da Ta.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if ( Conn! = Null) {sendTo (conn, {type: "leave"});}}}}); Connection.send ("Olá mundo"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Aplicação de cliente. Uma maneira de testar este aplicativo está abrindo duas guias do navegador e tentando fazer uma ligação de áudio entre si. Em primeiro lugar, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da web. Você pode aprender mais em http://getbootstrap.com/. Crie uma pasta chamada, por exemplo, "audiochat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um pacote de arquivos.json (é necessário gerenciar dependências npm) e adicione o seguinte - {"nome": "webrtc-audiochat", "versão": "0.1.0", "descrição": "webrtc -audiochat "," autor ":" Autor "," licença ":" BSD-2-Clause "} Em seguida, execute o npm install bootstrap. Isso instalará a biblioteca bootstrap na pasta audiochat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Voice Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Voice Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <button id =" loginBtn "class =" btn btn-lg btn-primary btnblock "> Iniciar sessão </ button> </ div> </ div> </ div> < Div id = "callPage" class = "call-page"> <div class = "row"> <div class = "col-md-6 text-right"> áudio local: <audio id = "localAudio" controla a reprodução automática> </ Audio> </ div> <div class = "col-md-6 text-left"> Áudio remoto: <audio id = "remoteAudio" controla a reprodução automática> </ audio> </ div> </ div> <div Class = "row text-center"> <div class = "col-md-12"> <input id = "callToUsernameInput" type = "text" placeholder = "nome de usuário para ligar" /> <button id = classe "callBtn" = "Btn-sucesso btn"> Ligar </ button> <botão id = "hangUpBtn" class = "btn-danger btn"> Desligar </ button> </ div> </ div> </ div> <script src = "Client.js"> </ script> </ body> </ html> Esta página deve ser b É familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Você deve ver os dois audiO elementos para fluxos de áudio locais e remotos. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem sucedido, mostramos a página de chamada e começamos a configurar uma conexão de pares. Para iniciar uma conexão entre pares, precisamos - • Obter um fluxo de áudio de um microfone • Criar o objeto RTCPeerConnectionAdicione o seguinte código ao "bloco de seleção UI "-var localAudio = document.querySelector ('# localAudio'); Var remoteAudio = document.querySelector ('# remoteAudio'); Var yourConn; Var stream; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de áudio local navigator.webkitGetUserMedia ({video: falso, áudio: true}, função (myStream) {stream = myStream; // exibindo fluxo de áudio local na página localAudio.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteAudio.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate",});}};}, function (error) {console. Log (erro);}); }}; Agora, se você executar o código, a página deve permitir que você faça login e exiba seu fluxo de áudio local na página. Agora estamos prontos para iniciar uma chamada. Em primeiro lugar, enviamos uma oferta para outro usuário. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando uma chamada callBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // Crie uma oferta yourConn.createOffer (função (oferta) {send ({type: "offer", offer: offer}); yourConn.setLocalDescription (offer);}, function (error) {aLert ("Erro ao criar uma oferta"); }); }}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }: Adicionamos um manipulador de cliques ao botão Ligar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador onmessage. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O último passo é implementar o recurso de desligamento. Isso deixará de transmitir dados e informará ao outro usuário para fechar a chamada. Adicione o seguinte código - // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteAudio.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null;}; Quando o usuário clicar no botão Desligar - • Ele enviará uma mensagem "deixar" para o outro usuário • Ele fechará o RTCPeerConnection e destruirá a conexão localNada execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. Você pode então fazer uma ligação de áudio para a guia e desligar a chamada. O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var localAudio = document.querySelector ('# localAudio'); Var remoteAudio = document.querySelector ('# remoteAudio'); Var yourConn; Fluxo de var; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // recebendo fluxo de áudio local navigator.webkitGetUserMedia ({video: falso, áudio: true}, função (myStream) {stream = myStream; // exibindo fluxo de áudio local na página localAudio.src = window.URL. CreateObjectURL (stream); // usando o Google stun public var configuration var = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; yourConn = new webkitRTCPeerConnection (configuração) ; // setup stream escutando yourConn.addStream (stream); // quando um usuário remoto adiciona stream para a conexão do peer, nós o exibimos yourConn.onaddstream = function (e) {remoteAudio.src = window.URL.createObjectURL (e. Stream);}; // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate});}};}}, função ( Erro) {console.log (erro);}); }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {aleRt ("Erro ao criar uma oferta"); }); }}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // hang uphangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; RemoteAudio.src = null; YourConn.close (); YourConn.onicecandidate = null; YourConn.onaddstream = null; }; WebRTC - Demo de texto Neste capítulo, vamos criar um aplicativo cliente que permita que dois usuários em dispositivos separados enviem mensagens entre si usando o WebRTC. Nossa aplicação terá duas páginas. Um para o login e outro para enviar mensagens para outro usuário. As duas páginas serão as tags div. A maior parte da entrada é feita através de manipuladores de eventos simples.Signaling ServerPara criar uma conexão WebRTC, os clientes devem ser capazes de transferir mensagens sem usar uma conexão de ponto WebRTC. Aqui é onde vamos usar o HTML5 WebSockets - uma conexão de soquete bidirecional entre dois pontos finais - um servidor web e um navegador da Web. Agora vamos começar a usar a biblioteca do WebSocket. Crie o arquivo server.js e insira o seguinte código - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {console.log ("Obter mensagem de um usuário:", mensagem);}); connection.send ("Olá do servidor");}); A primeira linha requer a biblioteca do WebSocket que já temos instalado. Em seguida, criamos um servidor de soquetes na porta 9090. Em seguida, ouvimos o evento de conexão. Este código será executado quando um usuário fizer uma conexão WebSocket com o servidor. Em seguida, ouvimos as mensagens enviadas pelo usuário. Finalmente, enviamos uma resposta para o usuário conectado, dizendo "Olá do servidor". No nosso servidor de sinalização, usaremos um nome de usuário baseado em string para cada conexão para saber onde enviar mensagens. Vamos mudar o nosso manual de conexão um bit -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ( "JSON inválido"); data = {};}}); assim aceitamos apenas mensagens JSON. Em seguida, precisamos armazenar todos os usuários conectados em algum lugar. Usaremos um objeto Javascript simples para isso. Mude o topo do nosso arquivo - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; vamos adicionar um campo de tipo para cada mensagem proveniente do cliente. Por exemplo, se um usuário quiser entrar, ele envia a mensagem de tipo de login. Vamos defini-lo -connection.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido "); Data = {};} // tipo de comutação da mudança de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", data.name ); // se alguém estiver logado com esse nome de usuário, recusar se (usuários [data.name]) {sendTo (conexão, {digite: "login", sucesso: falso});} else {// salvar conexão do usuário em Os usuários do servidor [data.name] = connection; connection.name = data.name; sendTo (connection, {type: "login", success: true});} break; default: sendTo (connection, {type: "error ", Mensagem:" Comando não encontrado: "+ data.type}); break;}}); se o usuário envia uma mensagem com o tipo de login, nós - • Verifique se alguém já iniciou sessão com este nome de usuário. • Se Então, diga ao usuário que ele não iniciou sessão com sucesso. • Se ninguém estiver usando esse nome de usuário, adicionamos userna Eu como uma chave para o objeto de conexão. • Se um comando não for reconhecido, enviamos um erro. O código a seguir é uma função auxiliar para enviar mensagens para uma conexão. Adicione-o ao arquivo server.js -function sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } Quando o usuário desconecta, devemos limpar sua conexão. Podemos excluir o usuário quando o evento de fechamento é disparado. Adicione o seguinte código ao manipulador de conexão -connection.on ("fechar", function () {if (connection.name) {delete users [connection.name];}}); Após o login com êxito, o usuário deseja chamar outro. Ele deve fazer uma oferta para outro usuário para alcançá-lo. Adicione a oferta handler -case "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); Em primeiro lugar, temos a conexão do usuário que estamos tentando chamar. Se existe, nós lhe enviamos detalhes da oferta. Nós também adicionamos otherName ao objeto de conexão. Isso é feito para a simplicidade de encontrá-lo mais tarde. Responder à resposta tem um padrão similar que usamos no documentador. Nosso servidor simplesmente passa por todas as mensagens como resposta a outro usuário. Adicione o seguinte código após o handler da oferta - "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } Break; A parte final está lidando com candidatos ICE entre usuários. Usamos a mesma técnica apenas passando mensagens entre usuários. A principal diferença é que as mensagens candidatas podem acontecer várias vezes por usuário em qualquer ordem. Adicione o candidato handler -case "candidato": console.log ("Enviando candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } Break; Para permitir que nossos usuários se desconectem de outro usuário, devemos implementar a função de suspensão. Ele também informará o servidor para excluir todas as referências do usuário. Adicione o carrinho de licença - caixa "sair": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Isso também enviará ao outro usuário o evento de licença para que ele possa desconectar sua conexão entre pares de acordo. Também devemos lidar com o caso quando um usuário solta sua conexão do servidor de sinalização. Vamos modificar o nosso close handler -connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", conexão .otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); O seguinte é O código completo do nosso servidor de sinalização - // exige nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criação de um servidor websocket na porta 9090 var wss = new WebSocketServer ({port: 9090}); // todos conectados aos usuários do servidor var users = {}; // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {console.log ("Usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem' , Função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // Tipo de chave de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário conectado", data.name); // se alguém estiver logado com este nome de usuário Em seguida, recusar se (users [data.name]) {sendTo (connection, {type: "login", success: false});} else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; conexão .name = data.name; sendTo (connection, {type: "login", success: true});} break; case "offer": // para o ex. UserA quer chamar UserB console.log ("Enviando oferta para : ", Data.name); // se UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; if (conn! = Null) {/ / Configuração que UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {excluir usuários [connection.name]; se (connection.otherName) {console.log ("DiscoNnecting from ", connection.otherName); var conn = users [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type:" leave "});}}}}} ); Connection.send ("Olá mundo");}); função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem));} Aplicação de cliente. Uma maneira de testar esta aplicação está abrindo duas abas de navegador e tentando Para enviar uma mensagem. Em primeiro lugar, precisamos instalar a biblioteca de bootstrap. O Bootstrap é uma estrutura de frontend para o desenvolvimento de aplicativos da Web. Você pode aprender mais em http://getbootstrap.com/.Crear uma pasta chamada, por exemplo, , "Textchat". Esta será a nossa pasta de aplicação raiz. Dentro desta pasta, crie um pacote de arquivos.json (é necessário gerenciar dependências npm) e adicione o seguinte - {"name": "webrtc-textochat", "versão" : "0.1.0", "descrição": "webrtc-textchat", "autor": "Autor", "licença": "BSD-2-Clause"} Em seguida, execute npm install bootstrap. Isso instalará a biblioteca bootstrap em a Pasta textchat / node\_modules. Agora precisamos criar uma página HTML básica. Crie um arquivo index.html na pasta raiz com o seguinte código - <html> <head> <title> WebRTC Text Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap .min.css "/> </ head> <style> body {background: #eee; Estofamento: 5% 0; } </ Style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row"> <div class = "col-md-4 col-md-offset-4" > <H2> WebRTC Text Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <button id =" loginBtn "class =" btn btn-lg btn-primary btnblock "> Iniciar sessão </ button> </ div> </ div> </ div> < Div id = "callPage" class = "container de página de chamada"> <div class = "row"> <div class = "col-md-4 col-md-offset-4 text-center"> <div class = " Painel de painel-primário "> <div class =" painel-título "> bate-papo de texto </ div> <div id =" chatarea "class =" painel-corpo texto-esquerdo "> </ div> </ div> </ Div> </ div> <div class = "linha de texto do centro do formulário-grupo"> <div class = "col-md-12"> <input id = "callToUsernameInput" type = "text" placeholder = "nome de usuário para ligar "/> <Botão id =" callBtn "class =" btn-sucesso btn "> Chamar </ button> <botão id =" hangUpBtn "class =" btn-danger btn "> Desligar </ button> </ div> </ Div> <div class = "row text-center"> <div class = "col-md-12"> <input id = "msgInput" type = "Texto" espaço reservado = "mensagem" /> <botão id = "sendMsgBtn" class = "btn-sucesso btn"> Enviar </ button> </ div> </ div> </ div> <script src = "cliente. Js "> </ script> </ body> </ html> Esta página deve ser familiar para você. Adicionamos o arquivo css bootstrap. Nós também definimos duas páginas. Finalmente, criamos vários campos de texto e botões para obter informações do usuário. Na página "chat", você deve ver a etiqueta div com a identificação "chatarea" onde todas as nossas mensagens serão exibidas. Observe que nós adicionamos um link para um arquivo client.js. Agora, precisamos estabelecer uma conexão com o nosso servidor de sinalização. Crie o arquivo client.js na pasta raiz com o seguinte código - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); } Agora, execute o nosso servidor de sinalização através do servidor de nó. Então, dentro da pasta raiz, execute o comando estático e abra a página dentro do navegador. Você deve ver a seguinte saída da consola - A próxima etapa é implementar um usuário logar com um nome de usuário exclusivo. Nós simplesmente enviamos um nome de usuário para o servidor, o que, em seguida, diz-nos se é ou não tomado. Adicione o seguinte código ao seu arquivo client.js - // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); Var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*}}; Em primeiro lugar, selecionamos algumas referências aos elementos na página. Ocultamos a página de chamada. Então, adicionamos um ouvinte de eventos no botão de login. Quando o usuário clica, enviamos seu nome de usuário para o servidor. Finalmente, implementamos o retorno de chamada do handleLogin. Se o login foi bem-sucedido, mostramos a página de chamada, configuramos uma conexão entre pares e criamos um canal de dados. Para iniciar uma conexão entre pares com um canal de dados que precisamos - • Criar o objeto RTCPeerConnection • Criar um canal de dados dentro do nosso objeto RTCPeerConnectionAdd O seguinte código para o "bloco de seletores UI" -var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Var dataChannel; Modificar a função handleLogin -function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; Se o login foi bem-sucedido, o aplicativo cria o objeto RTCPeerConnection e o gerenciador onicecandidate da configuração, que envia todos os icecandidatos encontrados para o outro ponto. Ele também cria um dataChannel. Observe que ao criar o objeto RTCPeerConnection o segundo argumento no construtor opcional: [{RtpDataChannels: true}] é obrigatório se você estiver usando o Chrome ou o Opera. O próximo passo é criar uma oferta para o outro ponto. Uma vez que um usuário obtém a oferta, ele cria uma resposta e começa a negociar candidatos do ICE. Adicione o seguinte código ao arquivo client.js - // iniciando um callcallBtn.addEventListener ("clique", function () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // create Uma oferta yourConn.createOffer (função (oferta) {enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta") ;});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");});}; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }: Adicionamos um manipulador de cliques ao botão Ligar, que inicia uma oferta. Em seguida, implementamos vários manipuladores esperados pelo manipulador onmessage. Eles serão processados ​​de forma assíncrona até que ambos os usuários tenham feito uma conexão. O próximo passo é implementar o recurso de desligamento. Isso deixará de transmitir dados e informará ao outro usuário para fechar o canal de dados. Adicione o seguinte código - // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; } Quando o usuário clicar no botão Desligar - • Ele enviará uma mensagem "deixar" para o outro usuário. • Ele fechará o RTCPeerConnection e também o canal de dados. O último passo é enviar uma mensagem para outro ponto. Adicione o manipulador "clique" ao botão "enviar mensagem" - // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = Nome +":" + Val + "<br />"; // enviando uma mensagem para um dado de dados de banco conectado.hannel.send (val); MsgInput.value = ""; }); Agora execute o código. Você deve poder fazer logon no servidor usando duas abas do navegador. Você pode então configurar uma conexão entre pares para o outro usuário e enviar-lhe uma mensagem, bem como fechar o canal de dados clicando no botão "Desligar". O seguinte é todo o arquivo client.js - // nome do nosso nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('ws: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização");}; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Dados varCanal; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; if (name.length> 0) {send ({type: "login", name: name} );}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... tente um nome de usuário diferente"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; }; // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = nome + ":" + val + "<br />" ; // enviando uma mensagem para um dado de dados de pares conectadosChannel.send (val); msgInput.value = "";}); WebRTC - SecurityNeste capítulo, vamos adicionar recursos de segurança ao servidor de sinalização que criamos na seção "WebRTC Signaling "capítulo. Haverá dois aprimoramentos - • Autenticação de usuário usando o banco de dados Redis • Ativando a conexão de soquete seguroFirstly, você deve instalar o Redis. • Baixe a versão estável mais recente em http://redis.io/download(3.05 no meu caso) • Descompactar • Interior A pasta de download executada sudo make install • Após a conclusão da instalação, execute make test para verificar se tudo está funcionando corretamente. Redis possui dois comandos executáveis ​​- • redis-cli - interface de linha de comando para Redis (parte do cliente) • redis-server - Redis data storePara executar o redis server type redis-server no console do terminal. Você deve ver o seguinte: agora abra uma nova janela do terminal e execute o redis-cli para abrir uma aplicação cliente. Basicamente, a Redis é uma base de dados de valores-chave. Para criar uma chave com um valor de seqüência de caracteres, você deve usar o comando SET. Para ler o valor da chave, você deve usar o comando GET. Vamos adicionar dois usuários e senhas para eles. As chaves serão os nomes de usuário e os valores dessas chaves serão as senhas correspondentes. Agora, devemos modificar nosso servidor de sinalização para adicionar uma autenticação de usuário. Adicione o seguinte código à parte superior do arquivo server.js - // requer a biblioteca redis no Node.js var redis = require ("redis"); // criando o objeto redis client var redisClient = redis.createClient (); no código acima, exigimos a biblioteca Redis para Node.js e criamos um cliente redis para o nosso servidor. Para adicionar a autenticação, modifique o manipulador de mensagens na conexão Objeto - // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on (' Mensagem ', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console.log ("JSON inválido"); data = {};} // verifique se um usuário é autenticado se (data.type! = "Login") {// se o usuário não for autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", mensagem: "You Não são autenticados "}); return;}} // tipo de comutação da mudança de mensagem do usuário (data.type) {// quando um usuário tenta acessar o caso" login ": console.log (" Usuário logado: ", dados .name); // obter senha para este nome de usuário de re Esta base de dados redisClient.get (data.name, function (err, reply) {// verifique se a senha coincide com a armazenada em redis var loginSuccess = reply === data.password; // se alguém estiver logado com esse nome de usuário ou senha incorreta, recuse se (usuários [data.name] ||! LoginSuccess) {sendTo (connection, {type: "login", success: false}); } Else {// salvar conexão do usuário nos usuários do servidor [data.name] = conexão; Connection.name = data.name; Connection.isAuth = true; SendTo (conexão, {digite: "login", sucesso: true}); }}); pausa; }}); } // ... // \*\*\*\*\* outros manipuladores \*\*\*\*\*\*\* No código acima, se um usuário tentar entrar, obtemos da Redis sua senha, verifique se ele coincide com o armazenado, e se É bem sucedido que nós armazenamos seu nome de usuário no servidor. Também adicionamos a bandeira isAuth à conexão para verificar se o usuário está autenticado. Observe este código - // verifique se um usuário é autenticado se (data.type! = "Login") {// se o usuário não está autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", Mensagem: "Você não está autenticado"}); Retorna; }} Se um usuário não autenticado tentar enviar uma oferta ou deixar a conexão, simplesmente enviamos um erro para trás. O próximo passo é habilitar uma conexão de soquete segura. É altamente recomendado para aplicações WebRTC. PKI (Public Key Infrastructure) é uma assinatura digital de uma autoridade de certificação (Certificate Authority). Os usuários então verificam se a chave privada usada para assinar um certificado corresponde à chave pública do certificado da CA. Para fins de desenvolvimento. Nós usaremos um certificado de segurança auto-assinado. Usamos o openssl. É uma ferramenta de código aberto que implementa protocolos SSL (Secure Sockets Layer) e TLS (Transport Layer Security). Geralmente é instalado por padrão em sistemas Unix. Execute a versão openssl -a para verificar se ela está instalada. Para gerar chaves de certificado de segurança públicas e privadas, você deve seguir as etapas abaixo: • Gerar uma senha temporária do servidor keyopenssl genrsa -des3 -passout pass: x -out server.pass.key 2048 • Gerar um servidor privado keyopenssl rsa-passin pass : 12345 -in server.pass.key -out server.key • Gerar uma solicitação de assinatura. Serão feitas perguntas adicionais sobre sua empresa. Basta pressionar o botão "Enter" todo o tempo.openssl req -new -key server.key -out server.csr • Gerar o certificateopenssl x509 -req -days 1095 -in server.csr -signkey server.key -out server.crt Agora você tem dois arquivos, o certificado (server.crt) e a chave privada (server.key). Copie-os para a pasta de raiz do servidor de sinalização. Para habilitar a conexão de soquete segura, modifique nosso servidor de sinalização. //require o módulo do sistema de arquivos var fs = require ('fs'); Var httpServ = require ('https'); //https://github.com/visionmedia/superagent/issues/205 process.env.NODE\_TLS\_REJECT\_NÃO AUTORIZADO = "0"; // o servidor seguro irá ligar à porta 9090 var cfg = {port: 9090, ssl\_key: 'server.key', ssl\_cert: 'server.crt'}; // no caso de solicitação http apenas envie de volta "OK" var processRequest = função (req, res) {res.writeHead (200); Res.end ("OK"); }; // crie o nosso servidor com o var var habilitado por SSL = httpServ.createServer ({chave: fs.readFileSync (cfg.ssl\_key), cert: fs.readFileSync (cfg.ssl\_cert)}, processRequest) .listen (cfg.port); // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criando um servidor de websocket na porta 9090 var wss = new WebSocketServer ({server: app}); // todos conectados aos usuários do servidor var users = {}; // requer a biblioteca redis em Node.jsvar redis = require ("redis"); // criando o objeto redis client var redisClient = redis.createClient (); // quando um usuário se conecta ao nosso wss.on ('conexão', função (conexão) {//...ou outro código. No código acima, exigimos que a biblioteca fs leia chave e certificado privado, crie o objeto cfg com A porta de ligação e os caminhos para a chave e o certificado privados. Em seguida, criamos um servidor HTTPS com as nossas chaves juntamente com o servidor WebSocket na porta 9090. Agora abra https: // localhost: 9090 no Opera. Você deve ver o seguinte - Clique no botão "Continue de qualquer maneira". Você deve ver a mensagem "OK". Para testar nosso servidor de sinalização seguro, modificaremos o aplicativo de bate-papo que criamos no tutorial "WebRTC Text Demo". Só precisamos adicionar um campo de senha. É todo o arquivo index.html - <html> <head> <title> WebRTC Text Demo </ title> <link rel = "stylesheet" href = "node\_modules / bootstrap / dist / css / bootstrap.min.css" /> </ Head> <style> body {background: #eee; padding: 5% 0;} </ style> <body> <div id = "loginPage" class = "container text-center"> <div class = "row "> < Div class = "col-md-4 col-md-offset-4"> <h2> WebRTC Text Demo. Faça login </ h2> <label for = "usernameInput" class = "sr-only"> Login </ label> <input type = "email" id = "usernameInput" class = "form-control formgroup" placeholder = " Login "required =" "autofocus =" "> <input type =" text "id =" passwordInput "class =" form-control form-group "placeholder =" Password "required =" "autofocus =" "> <id do botão = "LoginBtn" class = "btn btn-lg btn-primary btnblock"> Inscreva-se </ button> </ div> </ div> </ div> <div id = "callPage" class = "contêiner da página de chamada" > <Div class = "row"> <div class = "col-md-4 col-md-offset-4 text-center"> <div class = "panel panel-primary"> <div class = "painel-título "> Chat de texto </ div> <div id =" chatarea "class =" painel-corpo texto-esquerdo "> </ div> </ div> </ div> </ div> <div class =" row text- Grupo-grupo central "> <div class =" col-md-12 "> <input id =" callToUsernameInput "type =" text "placeholder =" nome de usuário para ligar "/> <button id =" callBtn "class =" btn -success btn "> Chamar </ button> <botão id =" hangUpBtn "class =" btn-dan Ger btn ">> <div class =" col-md-12 "> <input id =" msgInput "type = "Texto" espaço reservado = "mensagem" /> <botão id = "sendMsgBtn" class = "btn-sucesso btn"> Enviar </ button> </ div> </ div> </ div> <script src = "cliente. Js "> </ script> </ body> </ html> Também precisamos habilitar uma conexão de soquete segura no arquivo client.js através desta linha var conn = new WebSocket ('wss: // localhost: 9090'); . Observe o protocolo wss. Em seguida, o manipulador do botão de login deve ser modificado para enviar a senha juntamente com o nome de usuário -loginBtn.addEventListener ("clique", função (evento) {nome = nome\_de\_usuário.valor; var pwd = passwordInput.value; if (name.length> 0) { Enviar ({tipo: "login", nome: nome, senha: pwd});}}); O seguinte é o arquivo client.js inteiro - // nosso nome de nome de usuário; Var connectedUser; // conectando-se ao nosso servidor de sinalização var conn = new WebSocket ('wss: // localhost: 9090'); Conn.onopen = function () {console.log ("Conectado ao servidor de sinalização"); }; // quando recebemos uma mensagem de um servidor de sinalização conn.onmessage = function (msg) {console.log ("Got message", msg.data); Var data = JSON.parse (msg.data); Switch (data.type) {case "login": handleLogin (data.success); pausa; // quando alguém quer nos chamar de "oferta" caseira: handleOffer (data.offer, data.name); pausa; Caso "resposta": handleAnswer (data.answer); pausa; // quando um membro remoto envia um candidato de gelo para nós caso "candidato": handleCandidate (data.candidate); pausa; Caso "deixar": handleLeave (); pausa; Padrão: intervalo; }}; Conn.onerror = function (err) {console.log ("Got error", err); }; // alias para enviar mensagens de codificação JSON enviada (mensagem) {// anexe o outro nome de usuário de pares a nossas mensagens se (connectedUser) {message.name = connectedUser; } Conn.send (JSON.stringify (mensagem)); }; // \*\*\*\*\*\* // UI selectors block // \*\*\*\*\*\* var loginPage = document.querySelector ('# loginPage'); Var usernameInput = document.querySelector ('# usernameInput'); Var passwordInput = document.querySelector ('# passwordInput'); Var loginBtn = document.querySelector ('# loginBtn'); Var callPage = document.querySelector ('# callPage'); Var callToUsernameInput = document.querySelector ('# callToUsernameInput'); var callBtn = document.querySelector ('# callBtn'); Var hangUpBtn = document.querySelector ('# hangUpBtn'); Var msgInput = document.querySelector ('# msgInput'); Var sendMsgBtn = document.querySelector ('# sendMsgBtn'); Var chatArea = document.querySelector ('# chatarea'); Var yourConn; Dados varCanal; CallPage.style.display = "none"; // Login quando o usuário clicar no botão loginBtn.addEventListener ("clique", função (evento) {name = usernameInput.value; var pwd = passwordInput.value; if (name.length> 0) {send ({type: " Login ", nome: nome, senha: pwd});}}); Function handleLogin (sucesso) {if (success === false) {alert ("Ooops ... nome de usuário ou senha incorretos"); } Else {loginPage.style.display = "none"; CallPage.style.display = "block"; // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Iniciando uma conexão entre pares // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\* // usando o servidor de stun público do Google var configuration = {"iceServers": [{"url": "stun: stun2.1.google.com: 19302"}]}; YourConn = new webkitRTCPeerConnection (configuração, {opcional: [{RtpDataChannels: true}]}); // Configuração de gerenciamento de gelo yourConn.onicecandidate = função (evento) {if (event.candidate) {send ({type: "candidate", candidate: event.candidate}); }}; // criação de data channel dataChannel = yourConn.createDataChannel ("channel1", {reliable: true}); DataChannel.onerror = function (error) {console.log ("Ooops ... error:", erro); }; // quando recebemos uma mensagem do outro par, exiba-a na tela dataChannel.onmessage = função (evento) {chatArea.innerHTML + = connectedUser + ":" + event.data + "<br />"; }; DataChannel.onclose = function () {console.log ("canal de dados está fechado"); }; }}; // iniciando uma chamada callBtn.addEventListener ("clique", função () {var callToUsername = callToUsernameInput.value; if (callToUsername.length> 0) {connectedUser = callToUsername; // crie uma oferta yourConn.createOffer (função (oferta) {Enviar ({tipo: "oferta", oferta: oferta}); yourConn.setLocalDescription (oferta);}, função (erro) {alerta ("Erro ao criar uma oferta");});}}); // quando alguém nos envia uma função de oferta handleOffer (oferta, nome) {connectedUser = name; YourConn.setRemoteDescription (novo RTCSessionDescription (oferta)); // crie uma resposta para uma oferta yourConn.createAnswer (função (resposta) {yourConn.setLocalDescription (resposta); envie ({digite: "answer", answer: answer});}, function (error) {alert ("Error Ao criar uma resposta ");}); }; // quando obtivemos uma resposta a partir de uma função de usuário remoto handleAnswer (answer) {yourConn.setRemoteDescription (novo RTCSessionDescription (resposta)); }; // quando obtivemos um candidato ao gelo a partir de uma função de usuário remota handleCandidate (candidato) {yourConn.addIceCandidate (novo RTCIceCandidate (candidato)); }; // desligue hangUpBtn.addEventListener ("clique", function () {send ({type: "leave"}); handleLeave ();}); Function handleLeave () {connectedUser = null; YourConn.close (); YourConn.onicecandidate = null; }; // quando o usuário clicar no botão "enviar mensagem" sendMsgBtn.addEventListener ("clique", função (evento) {var val = msgInput.value; chatArea.innerHTML + = nome + ":" + val + "<br />" ; // enviando uma mensagem para um dado de dados de pares conectadosChannel.send (val); msgInput.value = "";}); agora execute nosso servidor de sinalização seguro através do servidor de nó. Execute o nó estático dentro da pasta de demonstração de bate-papo modificada. Abra o localhost: 8080 em duas guias do navegador. Tente iniciar sessão. Lembre-se de que apenas "user1" com "password1" e "user2" com "password2" podem entrar. Em seguida, estabeleça o RTCPeerConnection (chame outro usuário) e tente enviar uma mensagem. O seguinte é o código completo do nosso servidor de sinalização seguro - // requer o módulo do sistema de arquivos var fs = require ('fs'); Var httpServ = require ('https'); // https: //github.com/visionmedia/superagent/issues/205 process.env.NODE\_TLS\_REJECT\_UNAUTHORIZED = "0"; // o servidor seguro irá ligar à porta 9090 var cfg = {port: 9090, ssl\_key: 'server.key', ssl\_cert: 'server.crt'}; // no caso de solicitação http apenas envie de volta "OK" var processRequest = função (req, res) {res.writeHead (200); Res.end ("OK"); }; // crie o nosso servidor com o var var habilitado por SSL = httpServ.createServer ({chave: fs.readFileSync (cfg.ssl\_key), cert: fs.readFileSync (cfg.ssl\_cert)}, processRequest) .listen (cfg.port); // requer nossa biblioteca de websocket var WebSocketServer = require ('ws'). Servidor; // criando um servidor de websocket na porta 9090 var wss = new WebSocketServer ({server: app}); // todos conectados aos usuários do servidor var users = {}; // requer a biblioteca redis em Node.js var redis = require ("redis"); // criando o objeto do cliente redis var redisClient = redis.createClient (); // quando um usuário se conecta ao nosso sever wss.on ('conexão', função (conexão) {console.log ("usuário conectado"); // Quando o servidor recebe uma mensagem de uma conexão de usuário conectada.on ('mensagem', função (mensagem) {var data; // aceitando apenas mensagens JSON tente {data = JSON.parse (mensagem);} catch (e) {console. Log ("JSON inválido"); data = {};} // verifique seUm usuário é autenticado se (data.type! = "Login") {// se o usuário não for autenticado se (! Connection.isAuth) {sendTo (connection, {type: "error", mensagem: "You are not authenticated" }); Retorna; }} // tipo de comutação do parâmetro de mensagem do usuário (tipo de dados) {// quando um usuário tenta acessar o caso "login": console.log ("Usuário logado:", data.name); // obter senha para este nome de usuário a partir do banco de dados redis redisClient.get (data.name, function (err, reply) {// verificar se a senha coincide com a armazenada em redis var loginSuccess = reply === data.password; // Se alguém estiver logado com esse nome de usuário ou senha incorreta, recusar se (usuários [data.name] ||! LoginSuccess) {sendTo (connection, {type: "login", success: false}); else else // // Conexão do usuário nos usuários do servidor [data.name] = conexão; connection.name = data.name; connection.isAuth = true; sendTo (conexão, {digite: "login", sucesso: true});}}); pausa; Caso "oferta": // por ex. UserA quer chamar UserB console.log ("Enviando oferta para:", data.name); // se o UserB existir, envie-lhe os detalhes da oferta var conn = users [data.name]; Se (conn! = Null) {// configurando o UserA conectado com UserB connection.otherName = data.name; SendTo (conn, {type: "offer", oferta: data.offer, nome: connection.name}); } pausa; Caso "resposta": console.log ("Enviando resposta para:", data.name); // por ex. UserB responde UserA var conn = users [data.name]; Se (conn! = Null) {connection.otherName = data.name; SendTo (conn, {type: "answer", answer: data.answer}); } pausa; Caso "candidato": console.log ("Enviar candidato para:", data.name); Var conn = users [data.name]; If (conn! = Null) {sendTo (conn, {type: "candidato", candidato: data.candidate}); } pausa; Caso "deixar": console.log ("Desligar de", data.name); Var conn = users [data.name]; Conn.otherName = null; // notifica o outro usuário para que ele possa desconectar sua conexão de pares se (conn! = Null) {sendTo (conn, {type: "leave"}); } pausa; Connection.on ("close", function () {if (connection.name) {delete users [connection.name]; if (connection.otherName) {console.log ("Desconectando de", connection.otherName); = Usuários [connection.otherName]; conn.otherName = null; if (conn! = Null) {sendTo (conn, {type: "leave"});}}}}}); Padrão: sendTo (conexão, {type: "error", mensagem: "Comando não encontrado:" + data.type}); pausa; }}); // quando o usuário sai, por exemplo, fecha uma janela do navegador // isso pode ajudar se ainda estivéssemos em "oferta", "resposta" ou "candidato" estado connection.on ("fechar", função () {if (conexão. Nome) {apagar usuários [connection.name];}}); Connection.send ("Olá do servidor"); }); Função sendTo (conexão, mensagem) {connection.send (JSON.stringify (mensagem)); } SumárioNeste capítulo, adicionamos autenticação de usuário ao nosso servidor de sinalização. Nós também aprendemos a criar certificados SSL auto-assinados e usá-los no escopo de aplicativos WebRTC .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ WebRTC - Recursos Úteis \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Publicidades\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Página AnteriorPróxima \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Os seguintes recursos contêm informações adicionais sobre o WebRTC. Por favor, use-os para obter um conhecimento mais aprofundado sobre este tópico. Links úteis na WebRTC • WebRTC - Referência para WebRTC. • Referência WebRTC - Referência para WebRTC. • WebRTC Wiki - Referência da Wikipedia para WebRTC.Useful Books on WebRTC • • Para alistar Seu site nesta página, por favor, envie um e-mail para contact@tutorialspoint.com\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Discutir WebRTC\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Com a Comunicação em tempo real da Web (WebRTC), as aplicações web modernas podem facilmente transmitir conteúdo de áudio e vídeo para milhões de pessoas. Neste tutorial, explicamos como você pode usar o WebRTC para configurar conexões peer-to-peer para outros navegadores da Web com rapidez e facilidade .\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Glossário do Computador \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_AppletA aplicativo Java pequeno que é baixado por um navegador ActiveX ou habilitado para Java. Uma vez que foi baixado, o applet será executado no computador do usuário. Os applets comuns incluem calculadoras financeiras e programas de desenho na web. Software ApplicationComputer que executa uma tarefa ou conjunto de tarefas, como processamento de texto ou desenho. Os aplicativos também são chamados de programas. Código Econômico Americano para Intercâmbio de Informações, um sistema de codificação para converter caracteres de teclado e instruções para o código de número binário que o computador entende.BandwidthA capacidade de uma conexão em rede. A largura de banda determina a quantidade de dados que podem ser enviados ao longo dos fios em rede. A largura de banda é particularmente importante para conexões de Internet, uma vez que uma maior largura de banda também significa download mais rápido. Código binário A linguagem mais básica que um computador entendeS, é composto de uma série de 0s e 1s. O computador interpreta o código para formar números, letras, sinais de pontuação e símbolos. Por baixo, o menor número de informações do computador, seja o número 0 ou 1. Em suma, eles são chamados de dígitos binários. Para iniciar um computador. A inicialização a frio significa reiniciar o computador depois que o poder é desligado. A inicialização quente significa reiniciar o computador sem desligar o poder. BerserSoftware usado para navegar na Internet. O Google Chrome, o Firefox, o Netscape Navigator e o Microsoft Internet Explorer são os navegadores mais populares de hoje para acessar o mau funcionamento da World Wide Web.BugA devido a um erro no programa ou a um defeito nos computadores equipment.ByteMost usa combinações de oito bits, chamado bytes, Para representar um caractere de dados ou instruções. Por exemplo, a palavra gato tem três caracteres e seria representada por três bytes.CacheA pequena área de armazenamento de memória de dados que um computador pode usar para acessar novamente dados instantaneamente em vez de reler os dados da fonte original, como Como um disco rígido. Os navegadores usam um cache para armazenar páginas da web para que o usuário possa vê-los novamente sem se reconectar ao Web.CAD-CAMComputador Aided Drawing - Computer Aided Manufacturing. As instruções armazenadas em um computador que serão traduzidas para instruções de operação muito precisas para um robô, como a montagem de carros ou a sinalização de corte a laser. Memória de leitura de CD-ROMCompact Disc, um disco de leitura óptica projetado para armazenar informações como música , Materiais de referência ou software de computador. Um único CD-ROM pode conter cerca de 640 megabytes de dados, o suficiente para várias enciclopédias. A maioria dos programas de software agora são entregues em CD-ROM. A interface do Gateway CGICommon, um padrão de programação que permite que os visitantes preencham campos de formulário em uma página da Web e que essas informações interajam com um banco de dados, possivelmente retornando ao usuário como outra página da Web. O CGI também pode se referir a Criação de imagens geradas por computador, o processo em que programas de computador sofisticados criam gráficos imóveis e animados, como efeitos especiais para filmes.ChatTipando texto em uma caixa de mensagem em uma tela para dialogar com uma ou mais pessoas através do Internet ou outra rede.ChipA minúsculo wafer de silício que contém circuitos elétricos em miniatura que podem armazenar milhões de bits de informação.ClientUm usuário único de uma aplicação de rede que é operado a partir de um servidor. Uma arquitetura cliente / servidor permite que muitas pessoas usem os mesmos dados simultaneamente. O componente principal do programa (os dados) reside em um servidor centralizado, com componentes menores (interface do usuário) em cada cliente.Cookie Um arquivo de texto enviado por um servidor da Web que é armazenado no disco rígido de um computador e retorna ao servidor da Web Coisas sobre o usuário, seu computador e / ou suas atividades de computador. Unidade de processamento de CPUCentral. O cérebro do computador.CrackerA pessoa que invade um computador através de uma rede, sem autorização e com intenção maliciosa ou destrutiva. Problema de hardware ou software da CrashA que faz com que as informações sejam perdidas ou que o computador funcione mal. Às vezes, um acidente pode causar danos permanentes a um computador.Cursor Um indicador de posição móvel exibido em um monitor de computador que mostra um operador de computador onde a próxima ação ou operação ocorrerá.CyberspaceSlang para internet ou seja. Um conglomerado internacional de redes informáticas interconectadas. Iniciado no final da década de 1960, foi desenvolvido na década de 1970 para permitir que pesquisadores do governo e da universidade compartilhem informações. A Internet não é controlada por nenhum grupo ou organização. Seu foco original foi pesquisa e comunicação, mas continua expandindo, oferecendo uma ampla gama de recursos para empresas e usuários domésticos. Banco de dados Uma coleção de informações similares armazenadas em um arquivo, como um banco de dados de endereços. Essas informações podem ser criadas e armazenadas em um sistema de gerenciamento de banco de dados (DBMS) .DebugSlang. Para encontrar e corrigir defeitos do equipamento ou programar malfunções. Padrão A configuração pré-definida de um sistema ou aplicativo. Na maioria dos programas, os padrões podem ser alterados para refletir preferências pessoais. Desktop O diretório principal da interface do usuário. Os desktops geralmente contêm ícones que representam links para o disco rígido, uma rede (se houver) e um lixo ou lata de reciclagem para arquivos a serem excluídos. Ele também pode exibir ícones de aplicativos usados ​​com freqüência, conforme solicitado pelo usuário. Publicação do desktop. Produção de documentos de qualidade de publicação usando um computador pessoal em combinação com textos, gráficos e programas de layout de página.Directório Um repositório onde todos os arquivos são mantidos no computador. DiskTwo tipos distintos. Os nomes referem-se à mídia dentro do recipiente: um disco rígido armazena grandes quantidades de dados. É geralmente dentro do computador, mas pode ser um periférico separado no exterior. Os discos rígidos são compostos de vários discos de metal revestidos rígidos. Atualmente, os discos rígidos podem armazenar 15 a 30 Gb (gigabytes). Um disco flexível, 3,5 "quadrado, geralmente inserido no computadorE pode armazenar cerca de 1,4 megabytes de dados. Os disquetes quadrados de 3,5 "possuem um disco muito fino e flexível no interior. Também existe um disco flexível de tamanho intermediário, discos Zip com marca registrada, que pode armazenar 250 megabytes de dados. Unidade de disco. O equipamento que opera um disco rígido ou flexível.Domain representa um IP (Protocolo de Internet) ou conjunto de endereços IP que compõem um domínio. O nome do domínio aparece nos URLs para identificar páginas da web ou em endereços de e-mail. Por exemplo, o endereço de e-mail para a Primeira Dama é first.lady@whitehouse.gov, whitehouse .gov, sendo o nome de domínio. Cada nome de domínio termina com um sufixo que indica o domínio de nível superior ao qual pertence. Estes são: .com para comercial, .gov para governo, .org para organização, .edu para instituição educacional,. Biz para negócios, .info para informações, .tv para televisão, .ws para o site. Os sufixos de domínio também podem indicar o país no qual o domínio está registrado. Nenhuma das duas partes pode possuir o mesmo nome de domínio. Nome antigoO nome de uma rede Ou computador ligado ao Int Ernet. Os domínios são definidos por um endereço IP comum ou um conjunto de endereços IP (Internet Protocol) semelhantes. Download do processo de transferência de informações de um site (ou outro local remoto em uma rede) para o computador. É possível baixar um arquivo que inclui texto, imagem, áudio, vídeo e muitos outros.DOSDisk sistema operacional. Um sistema operacional projetado para PCs iniciais IBM compatíveis. Menu decrescenteA janela de menu que se abre verticalmente na tela para exibir opções relacionadas ao contexto. Também chamado de menu pop-up ou menu suspenso.Linha de assinante do SDLDigital, um método de conexão à Internet através de uma linha telefônica. Uma conexão DSL usa linhas telefônicas de cobre, mas é capaz de retransmitir dados a velocidades muito maiores do que os modems e não interfere com o uso do telefone. Disco Dvd Digital. Semelhante a um CD-ROM, ele armazena e reproduz áudio e vídeo.E-bookUn dispositivo eletrônico de leitura eletrônico (geralmente de mão) que permite que uma pessoa visualize materiais de leitura armazenados digitalmente. Email eletrônico; Mensagens, incluindo memorandos ou cartas, enviadas eletronicamente entre computadores em rede que podem estar em todo o escritório ou em todo o mundo.EmoticonA expressão baseada em texto de emoção criada a partir de caracteres ASCII que imita uma expressão facial quando visualizada com a cabeça inclinada para a esquerda. Aqui estão alguns exemplos: • Sorrindo • Franzir o pé • Piscadela • CryingEncryptionO processo de transmissão de dados codificados para que apenas os destinatários autorizados possam desembaralá-lo. Por exemplo, a criptografia é usada para codificar as informações do cartão de crédito quando as compras são feitas através da Internet.EthernetA tipo de rede.Ethernet cardA placa dentro de um computador ao qual um cabo de rede pode ser anexado.File Um conjunto de dados que está armazenado no computador. Firewall Um conjunto de programas de segurança que protegem um computador de interferências externas ou acessam através da estrutura Internet.FolderA para conter arquivos eletrônicos. Em alguns sistemas operacionais, ele é chamado de diretório.FontsSets de tipos de letra (ou caracteres) que vêm em diferentes estilos e tamanhos. FreewareSoftware criado por pessoas que estão dispostas a distribuí-lo para a satisfação de compartilhar ou saber que ajudaram a simplificar as pessoas vidas. Pode ser um software independente, ou pode adicionar funcionalidades ao software existente. Protocolo de Transferência de Arquivos FTP, um formato e um conjunto de regras para transferir arquivos de um host para um computador remoto.Gigabyte (GB) 1024 megabytes. Também chamado de gig.Glitch A causa de um mau funcionamento inesperado. Ferramenta de pesquisa avançada da Internet que permite aos usuários acessar informações textuais através de uma série de menus ou se estiver usando FTP, através de downloads. Interface de usuário genérica, um sistema que simplifica a seleção de comandos do computador, permitindo a O usuário aponta para símbolos ou ilustrações (chamados ícones) na tela do computador com um mouse.GroupwareSoftware que permite que pessoas em rede formem grupos e colaborem em documentos, programas ou bases de dados. Hacker Uma pessoa com experiência técnica que experimenta com sistemas de computador para determinar como Para desenvolver recursos adicionais. Os hackers ocasionalmente são solicitados pelos administradores do sistema a tentar entrar nos sistemas através de uma rede para testar a segurança. O termo hacker às vezes é incorretamente usado de forma intercambiável com cracker. Um hacker é chamado de um chapéu branco e um biscoito de um chapéu preto.Captura horáriaEm impressa em papel do que você preparou no computador.Hard drive Outro nome para o disco rígido que armazena informações em um computador.HardwareEs componentes físicos e mecânicos de um sistema de computador , Como o circuito eletrônico, chips, monitor, discos, unidades de disco, teclado, modem e impressora. Página inicial A página principal de um site usado para cumprimentar visitantes, fornecer informações sobre o site ou dirigir o visualizador para outras páginas No site.HTMLHypertext Markup Language, um padrão de convenções de marcação de texto usado para documentos na World Wide Web. Os navegadores interpretam os códigos para fornecer a estrutura de texto e a formatação (como negrito, azul ou itálico). HTTPHypertext Transfer Protocol, um sistema comum usado para solicitar e enviar documentos HTML na World Wide Web. É a primeira parcela de todos os endereços de URL na World Wide Web.HTTPSHytextext Transfer Protocol Secure, geralmente usado em sites de internet intracompanhia. As senhas são necessárias para obter acesso.HyperlinkText ou uma imagem que está conectada por codificação de hipertexto a um local diferente. Ao selecionar o texto ou a imagem com um mouse, o computador salta para (ou exibe) o texto vinculado. HyppermediaIntégrar áudio, gráficos e / ou vídeo através de links incorporados no programa principal. Hípertext Um sistema para organizar texto através de links, em oposição a Uma hierarquia orientada por menus, como Gopher. A maioria das páginas da Web inclui links de hipertexto para outras páginas desse site ou para outros sites na World Wide Web.IconsSímbolos ou ilustrações que aparecem na tela do computador que indicam arquivos de programa ou outras funções do computador. InputData que entra em um dispositivo de computador. Dispositivo de entradaA Dispositivo, como um teclado, uma caneta e um tablet, um mouse, um disco ou um microfone, que permite a entrada de informações (letras, números, som, vídeo) para um computador. Mensagens instantâneas (IM) Um aplicativo de bate-papo que permite que duas ou mais pessoas Para se comunicar através da Internet através de mensagens digitadas em tempo real.Interface As interconexões que permitem que um dispositivo, um programa ou uma pessoa interajam. As interfaces de hardware são os cabos que conectam o dispositivo à sua fonte de energia e a outros dispositivos. As interfaces de software permitem que o programa se comunique com outros programas (como o sistema operacional) e as interfaces de usuário permitem que o usuário se comunique com o programa (por exemplo, via mouse, comandos de menu, ícones, comandos de voz, etc.). InternetUm conglomerado internacional De redes informáticas interconectadas. Iniciado no final da década de 1960, foi desenvolvido na década de 1970 para permitir que pesquisadores do governo e da universidade compartilhem informações. A Internet não é controlada por nenhum grupo ou organização. Seu foco original foi a pesquisa e as comunicações, mas continua a expandir, oferecendo uma ampla gama de recursos para empresas e usuários domésticos. Endereço IP (protocolo Internet) O endereço do protocolo Internet é um conjunto exclusivo de números usados ​​para localizar outro computador em uma rede. O formato de um endereço IP é uma sequência de quatro bits de quatro bits separada por períodos. Cada número pode ser de 0 a 255 (i.e., 1.154.10.255). Dentro de uma rede fechada, os endereços IP podem ser atribuídos aleatoriamente, no entanto, os endereços IP dos servidores web devem ser registrados para evitar duplicatas. Java Uma linguagem de programação orientada a objetos projetada especificamente para programas (particularmente multimídia) para serem usados ​​pela Internet. O Java permite que os programadores criem pequenos programas ou aplicativos (applets) para aprimorar sites. Script de Java / ECMA. Linguagem de programação usada quase que exclusivamente para manipular conteúdo em uma página da Web. As funções comuns de JavaScript incluem validação de formulários em uma página da Web, criação de menus dinâmicos de navegação de páginas e rollovers de imagens. Kilobyte (K ou KB) Igual a 1.024 bytes.LinuxA UNIX - como sistema operacional de código aberto desenvolvido principalmente por Linus Torvalds. O Linux é gratuito e funciona em muitas plataformas, incluindo PCs e Macintoshes. O Linux é um sistema operacional de código aberto, o que significa que o código-fonte do sistema operacional está disponível gratuitamente para o público. Os programadores podem redistribuir e modificar o código, desde que não coletem royalties em seu trabalho ou negam o acesso ao seu código. Uma vez que o desenvolvimento não se restringe a uma única corporação, mais programadores podem depurar e melhorar o código fonte mais rápido.Paptop e notebooks. Pequenos, leves computadores portáteis com bateria que podem caber no seu colo. Cada um tem uma tela de exibição de cristal líquido fina e plana.MacroA script que opera uma série de comandos para executar uma função. É configurado para automatizar tarefas repetitivas. Sistema operacional OSA Maci com uma interface gráfica de usuário, desenvolvida pela Apple para computadores Macintosh. Current System X.1. (10) combina a interface Mac tradicional com um forte UNIX subjacente. Sistema operacional para maior desempenho e estabilidade. Megabyte (MB) Igual a 1.048.576 bytes, geralmente arredondado para um milhão de bytes (também chamado de meg) .MemóriaTemporário de armazenamento para informações, incluindo aplicativos e documentos. As informações devem ser armazenadas em um dispositivo permanente, como um disco rígido ou CD-ROM antes que a energia seja desligada, ou as informações serão perdidas. A memória do computador é medida em termos da quantidade de informação que pode armazenar, comumente em megabytes ou gigabytes.MenuA lista de opções relacionadas ao contexto que os usuários podem escolher. Barra de menu A faixa horizontal na parte superior da janela de um aplicativo. Cada palavra na faixa possui um menu suspenso sensível ao contexto que contém recursos e ações que estão disponíveis para o aplicativo em uso. Experimente combinar dois ou mais arquivos em um único arquivo.MHzUma abreviatura para Megahertz, ou um milhão de hertz. Um MHz representa um milhão de ciclos de relógio por segundo e é a medida deVelocidade do computador com um microprocessador. Por exemplo, um microprocessador que funciona a 300 MHz executa 300 milhões de ciclos por segundo. Cada instrução que um computador recebe leva um número fixo de ciclos de relógio para realizar, portanto, quanto mais ciclos um computador pode executar por segundo, mais rápido seus programas serão executados. Megahertz também é uma unidade de medida para largura de banda.MicroprocessadorUma unidade de processamento central completa (CPU) contida em um único chip de silício. Minimizarum termo usado em um sistema operacional GUI que usa o Windows. Refere-se a reduzir uma janela para um ícone ou um rótulo na parte inferior da tela, permitindo que outra janela seja visualizada.ModemA dispositivo que conecta dois computadores em uma linha de telefone ou cabo convertendo os dados do computador em um sinal de áudio. O modem é uma contração para o processo que executa: modulate-demodulate.MonitorA terminal de exibição de vídeo.Mouse Um pequeno dispositivo portátil, semelhante a uma trackball, usado para controlar a posição do cursor na exibição de vídeo; Os movimentos do mouse em uma área de trabalho correspondem aos movimentos do cursor na tela. Formato de arquivo de áudio e vídeo MP3Compact. O tamanho pequeno dos arquivos facilita o download e o e-mail. Formato usado em dispositivos de reprodução portáteis. Programas de software multimídia que combinam texto e gráficos com som, vídeo e animação. Um PC multimídia contém o hardware para suportar esses recursos. Sistema operacional inicial MS-DOSAn desenvolvido pela Microsoft Corporation (Microsoft Disc Operating System) .NetworkA sistema de computadores interligados.Open sourceProgramas de computador cujo código-fonte original foi revelado ao público em geral para que ele Poderia ser desenvolvido abertamente. O software licenciado como código aberto pode ser alterado ou adaptado para novos usos, o que significa que o código-fonte do sistema operacional está disponível gratuitamente para o público. Os programadores podem redistribuir e modificar o código, desde que não coletem royalties em seu trabalho ou negam o acesso ao seu código. Uma vez que o desenvolvimento não se restringe a uma única empresa, mais programadores podem depurar e melhorar o código-fonte mais rápido. Sistema operacional Um conjunto de instruções que informam um computador sobre como operar quando está ligado. Ele configura um sistema de arquivamento para armazenar arquivos e informa ao computador como exibir informações em uma exibição de vídeo. A maioria dos sistemas operacionais de PCs são sistemas DOS (sistemas operados por disco), o que significa que as instruções são armazenadas em um disco (em oposição a serem originalmente armazenadas nos microprocessadores do computador). Outros sistemas operacionais bem conhecidos incluem UNIX, Linux, Macintosh e Windows.OutputData que saem de um dispositivo de computador. Por exemplo, informações exibidas no monitor, som dos alto-falantes e informações impressas em papel.PalmA computador portátil. Computador pessoal. Geralmente refere-se a computadores que executam o Windows com um processador Pentium. Placa de PC Placa de circuito impresso, uma placa impressa ou gravada com um circuito e processadores. Fontes de alimentação, dispositivos de armazenamento de informações ou trocadores estão anexados. Assistente digital pessoal da PD, um computador portátil que pode armazenar compromissos diários, números de telefone, endereços e outras informações importantes. A maioria dos PDAs se vincula a um computador desktop ou laptop para baixar ou fazer upload de informações. Formato de documento PDF, um formato apresentado por Adobe Acrobat que permite que os documentos sejam compartilhados em uma variedade de sistemas operacionais. Os documentos podem conter palavras e imagens e ser formatados para ter links eletrônicos para outras partes do documento ou para lugares na web.Pentium chipIntel da quinta geração de microprocessadores sofisticados de alta velocidade. Pentium significa o quinto elemento. Dispositivo externo de qualquer dispositivo conectado a um computador para melhorar a operação. Os exemplos incluem disco rígido externo, scanner, impressora, alto-falantes, teclado, mouse, trackball, stylus e tablet e joystick. Computador pessoal (PC) Um computador de usuário único contendo uma unidade de processamento central (CPU) e um ou mais circuitos de memória. PetabyteA medida de memória ou capacidade de armazenamento e é aproximadamente mil terabytes.PetaflopA medida teórica da velocidade de um computador e pode ser expressa como mil trilhões de operações de ponto flutuante por segundo.Plataforma O sistema operacional, como UNIX, Macintosh, Windows, em Com base em um computador. Conecte e toque no hardware ou nos periféricos do computador que são configurados com o software necessário, de modo que quando estão conectados a um computador, eles são reconhecidos pelo computador e estão prontos para usar. Menu inicialA janela de menu que se abre verticalmente ou horizontalmente Na tela para exibir opções relacionadas ao contexto. Também chamado de menu suspenso ou menu suspenso.Power PCA concorrente do chip Pentium. É uma nova geração de microprocessadores sofisticados e poderosos produzidos a partir de uma aliança Apple-IBM-Motorola. ImpressoraUm dispositivo mecânico para imprimir a saída de um computador no papel. Existem três tipos principais de impressoras: • Matriz de pontos: cria letras individuais, constituídas por uma série de pequenos pontos de tinta, perfurando uma fita com as extremidades dos pequenos fios. (Este tipo de printeR é usado com mais freqüência em ambientes industriais, como mala direta para rotulagem.) • Jato de tinta - pulveriza pequenas gotas de partículas de tinta no papel. • Laser - usa um feixe de luz para reproduzir a imagem de cada página usando uma carga magnética que Atrai toner seco que é transferido para papel e selado com calor. Programação Uma série precisa de instruções escritas em uma linguagem informática que informa ao computador o que fazer e como fazê-lo. Os programas também são chamados de software ou aplicações. Linguagem de programação Uma série de instruções escritas por um programador de acordo com um determinado conjunto de regras ou convenções (sintaxe). As linguagens de programação de alto nível são independentes do dispositivo no qual o aplicativo (ou programa) eventualmente será executado; Linguagens de baixo nível são específicas para cada programa ou plataforma. As instruções de linguagem de programação são convertidas em programas em linguagem específica para uma determinada máquina ou sistema operacional (linguagem de máquina). Para que o computador possa interpretar e executar as instruções. Algumas linguagens de programação comuns são BASIC, C, C ++, dBASE, FORTRAN e dispositivo de entrada Perl.PuckAn, como um mouse. Tem uma lupa com mira na frente dela que permite ao operador posicioná-lo precisamente ao traçar um desenho para uso com o software CAD-CAM. Menu suspensoA janela de menu que se abre verticalmente na tela para exibir opções relacionadas ao contexto . Também é chamado de menu suspenso ou menu pop-up. Tecnologia de opçõesInternet ferramenta que fornece informações específicas diretamente na área de trabalho do usuário, eliminando a necessidade de navegar por isso. O PointCast, que oferece novidades em categorias definidas pelo usuário, é um exemplo popular desta tecnologia. Software rápido e visual que permite a entrega de filmes via internet e e-mail. As imagens do QuickTime são visualizadas em um monitor. RAIDRedundant Array of Displays baratos, um método de divulgação de informações em vários discos configurados para atuar como uma unidade, usando duas técnicas diferentes: • Discos de disco - armazenando um pouco de informação em vários discos (em vez de Armazenando tudo em um disco e esperando que o disco não ocorra). • Espelhamento de disco - armazenando simultaneamente uma cópia de informações em outro disco para que as informações possam ser recuperadas se o disco principal falhar. Memória de acesso RAMRandom, uma das duas Tipos básicos de memória. Porções de programas são armazenadas na RAM quando o programa é iniciado para que o programa seja executado mais rápido. Embora um PC tenha uma quantidade fixa de RAM, apenas partes do mesmo serão acessadas pelo computador a qualquer momento. Também chamado de memória. Clique com o botão direito do mouse no botão direito do mouse para abrir menus suspensos sensíveis ao contexto.ROMRead-Only Memory, um dos dois tipos básicos de memória. ROM contém apenas informações permanentes colocadas pelo fabricante. As informações na ROM não podem ser alteradas, nem a memória pode ser alocada dinamicamente pelo computador ou pelo operador.Scanner. Um dispositivo eletrônico que usa equipamentos de detecção de luz para digitalizar imagens em papel, como texto, fotos e ilustrações e traduzir as imagens em sinais que o O computador pode então armazenar, modificar ou distribuir. Search EngineSoftware que permite pesquisar e recuperar material na Internet, particularmente na Web. Alguns motores de busca populares são o Alta Vista, o Google, o HotBot, o Yahoo !, Web Crawler e o Lycos.Server. Um computador que compartilha seus recursos e informações com outros computadores, chamados clientes, em uma rede.SharewareSoftware criado por pessoas que estão dispostas a vendê-lo A baixo custo ou nenhum custo para a gratificação do compartilhamento. Pode ser um software independente, ou pode adicionar funcionalidade a software existente. Programas de Software de Software; Também chamado de aplicativos. Os mecanismos de busca de processos do PiderA usam para investigar novas páginas em um site e coletam as informações que precisam ser colocadas em seus índices. Software de planilha que permite calcular números em um formato semelhante às páginas em um livro de contas convencional. StorageDevices usado para armazenar enormes quantidades de informações para que possa ser facilmente recuperado. Os dispositivos incluem RAIDs, CD-ROM, DVDs.StreamingTaking pacotes de informações (som ou visual) da Internet e armazenando-os em arquivos temporários para permitir que ele seja reproduzido em fluxo contínuo. Dispositivo de entrada de série e tableta semelhante a um mouse. A caneta é em forma de caneta. Ele é usado para desenhar em um tablet (como desenho em papel) eo tablet transfere as informações para o computador. O tablet responde à pressão. Quanto mais firme a pressão usada para desenhar, mais grossa é a linha. NavegaçãoExplorando o controle de Internet.Surge protectorA para proteger o computador e compensar variações na tensão.TelnetA maneira de se comunicar com um computador remoto por meio de um dispositivo network.TrackballInput que controla o Posição do cursor na tela; A unidade está montada perto do teclado e o movimento é controlado movendo uma bola. Tereabytes (TB) Mil gigabytes.TeraflopA medida da velocidade de um computador. Ele pode ser expresso como um trilhão de operações de ponto flutuante por segundo.Trojan HorseSee virus.UNIXA muito poderoso opSistema utilizado como base de muitas aplicações de computador high-end. Envolva o processo de transferência de informações de um computador para um site (ou outro local remoto em uma rede). Para transferir informações de um computador para um site (ou outro local remoto em uma rede) .URLUniform Resource Locator. • O protocolo para identificar um documento na Web. • Um endereço da Web (por exemplo, www.tutorialspoint.com). Um URL é exclusivo para cada usuário. Veja também o domínio.UPSUniversal Power Supply ou Ininterruptible Power Supply. Uma fonte de alimentação elétrica que inclui uma bateria para fornecer energia suficiente para um computador durante uma interrupção de dados de backup e desligar corretamente. O conector USB de soquete múltiplo de USB que permite que vários dispositivos compatíveis com USB sejam conectados a um computador. Quadro de boletim não modificado e não publicado na Internet que oferece milhares de fóruns, chamados newsgroups. Estes variam de grupos de notícias que trocam informações sobre avanços científicos para clubes de fãs de celebridades. Usuário amigável Um programa ou dispositivo cujo uso é intuitivo para pessoas com um fundo não técnico. Teleconferência por vídeo. Um "bate-papo cara a cara" remoto, quando duas ou mais pessoas usam Uma webcam e um bate-papo de conexão de telefone na Internet on-line. A webcam permite voz e vídeo ao vivo.Virtual reality (VR) Uma tecnologia que permite experimentar e interagir com imagens em um ambiente tridimensional simulado. Por exemplo, você poderia projetar uma sala em uma casa em seu computador e realmente sentir que você está caminhando por aí mesmo que nunca foi construído. (O Holodeck na série de televisão de ficção científica Star Trek: a Voyager seria a melhor realidade virtual.) A tecnologia atual exige que o usuário use um capacete especial, óculos de visão, luvas e outros equipamentos que transmitem e recebam informações do computador. Vírus Um pedaço não autorizado de código de computador conectado a um programa de computador ou partes de um sistema de computador que secretamente se copia de um computador para outro por discos compartilhados e por linhas de telefone e cabo. Pode destruir informações armazenadas no computador e, em casos extremos, pode destruir a operabilidade. Os computadores podem ser protegidos contra vírus se o operador utilizar um bom software de prevenção de vírus e manter as definições de vírus atualizadas. A maioria dos vírus não está programada para se espalhar. Eles precisam ser enviados para outro computador por e-mail, compartilhamento ou aplicativos. O worm é uma exceção, porque está programado para replicar-se enviando cópias para outros computadores listados no catálogo de endereços de e-mail no computador. Existem vários tipos de vírus, por exemplo: • Os vírus de inicialização colocam alguns de seus códigos no setor de disco de inicialização para serem executados automaticamente ao inicializar. Portanto, quando uma máquina infectada é inicializada, o vírus é carregado e executado. • Arquivar vírus anexados aos arquivos do programa (arquivos com a extensão .exe). Quando você executa o programa infectado, o código de vírus é executado. • Os vírus de macros copiam suas macros para modelos e / ou outros arquivos de documentos do aplicativo. • O Trojan Horse é um programa malicioso de segurança que está disfarçado como algo como um protetor de tela Ou jogo. • Worm lança um aplicativo que destrói informações em seu disco rígido. Ele também envia uma cópia do vírus a todos no caderno de endereços de e-mail do computador. Formato de som WAVA (onda pronunciada) usado para reproduzir sons em um computador. Câmera de vídeo / câmera de trabalho do WebcamA que leva imagens ao vivo e as envia para uma Web Browser.WindowA parte de uma tela de computador usada em uma interface gráfica que permite aos usuários selecionar comandos apontando para ilustrações ou símbolos com um mouse. "Windows" é também o nome adotado pela Microsoft para o seu sistema operacional popular. World Wide Web ("WWW" ou "Web") Uma rede de servidores na Internet que usa bancos de dados e arquivos com hipertexto. Foi desenvolvido em 1989 por Tim Berners-Lee, um cientista da computação britânica, e agora é a principal plataforma da Internet. O recurso que distingue a Web de outras aplicações da Internet é a sua capacidade de exibir gráficos, além de processador de texto. Processador de voz. Sistema ou programa de computador para configurar, editar, revisar, corrigir, armazenar e imprimir texto.WormVer vírus.WYSIWYGVocê vê é o que Você Obtém. Ao usar a maioria dos processadores de texto, os programas de layout de página (Consulte a publicação eletrônica) e os programas de design de páginas da Web, as palavras e as imagens serão exibidos no monitor, pois eles verão a página impressa ou a página da Web.