Using WebRTC with Django Channels HTMX & Coturn

Ken Whitesell – DjangoCon US 2024

Demo site

https://demo.kww.us/

- There is no separate user registration form
- Entering your first name and real name creates a user.
- No password is necessary.
  - The only requirement is that the first name matches the first name entered when the real name is entered

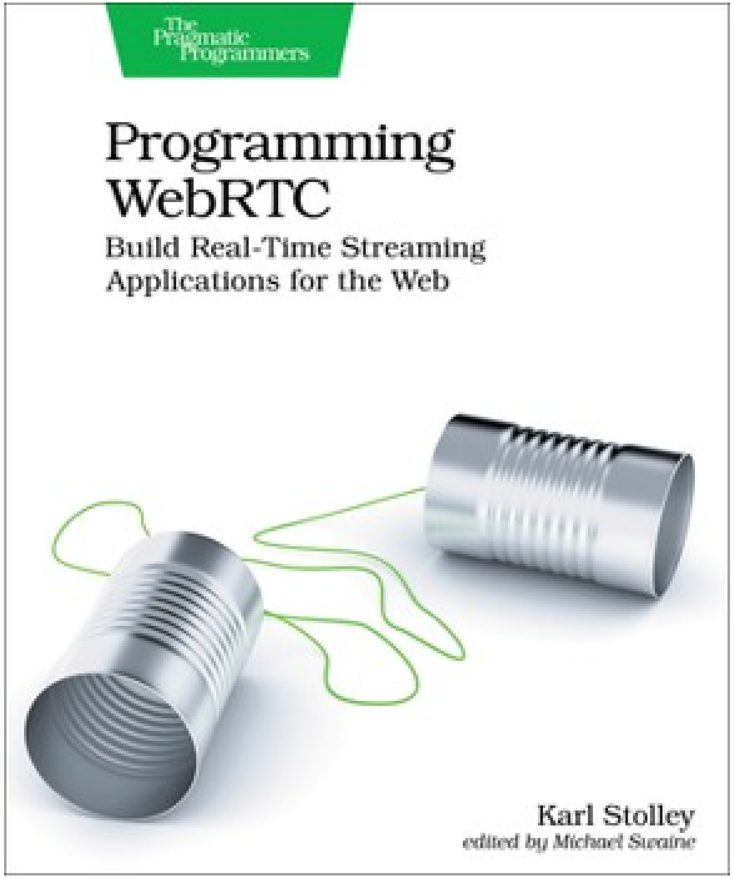
History

- WebRTC first released in 2011
- Standardized in 2018
- Adopted as W3C recommended standard in 2021
- Still tough to find current and accurate information

https://demo.kww.us/

History

- WebRTC first released in 2011
- Standardized in 2018
- Adopted as W3C recommended standard in 2021
- Still tough to find current and accurate information
- If you have interest in this topic, buy this book!

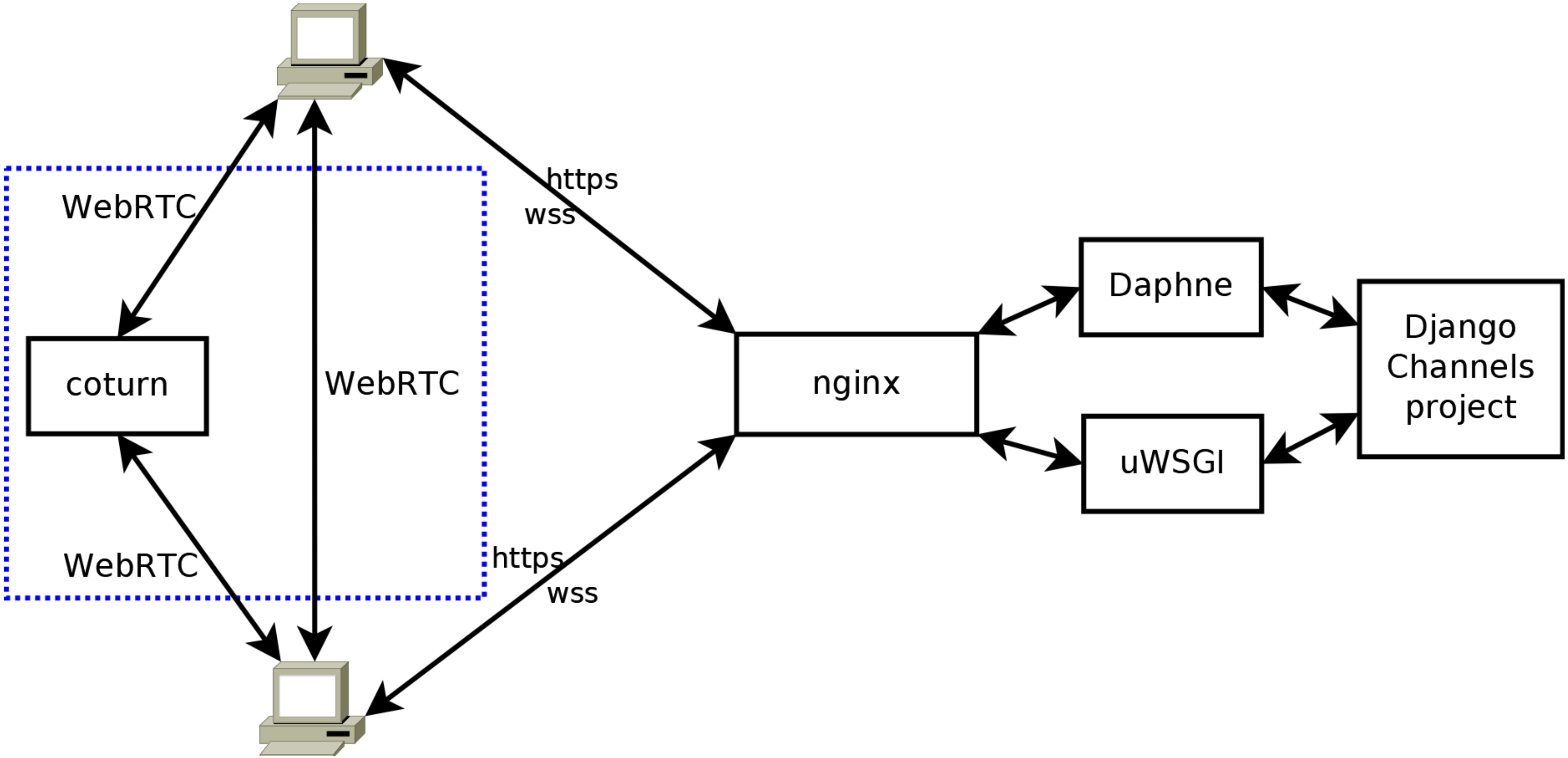


Demo site & Code – Final time

https://github.com/KenWhitesell/rtc_demo.git

https://demo.kww.us/

Overall architecture



The Parts

Django - https://docs.djangoproject.com/en/5.1/

- Serving web page and JavaScript, Provides authentication

Channels - https://github.com/django/channels

- Websocket for communications signaling

HTMX - https://htmx.org/

- Websocket in browser, HTML injection

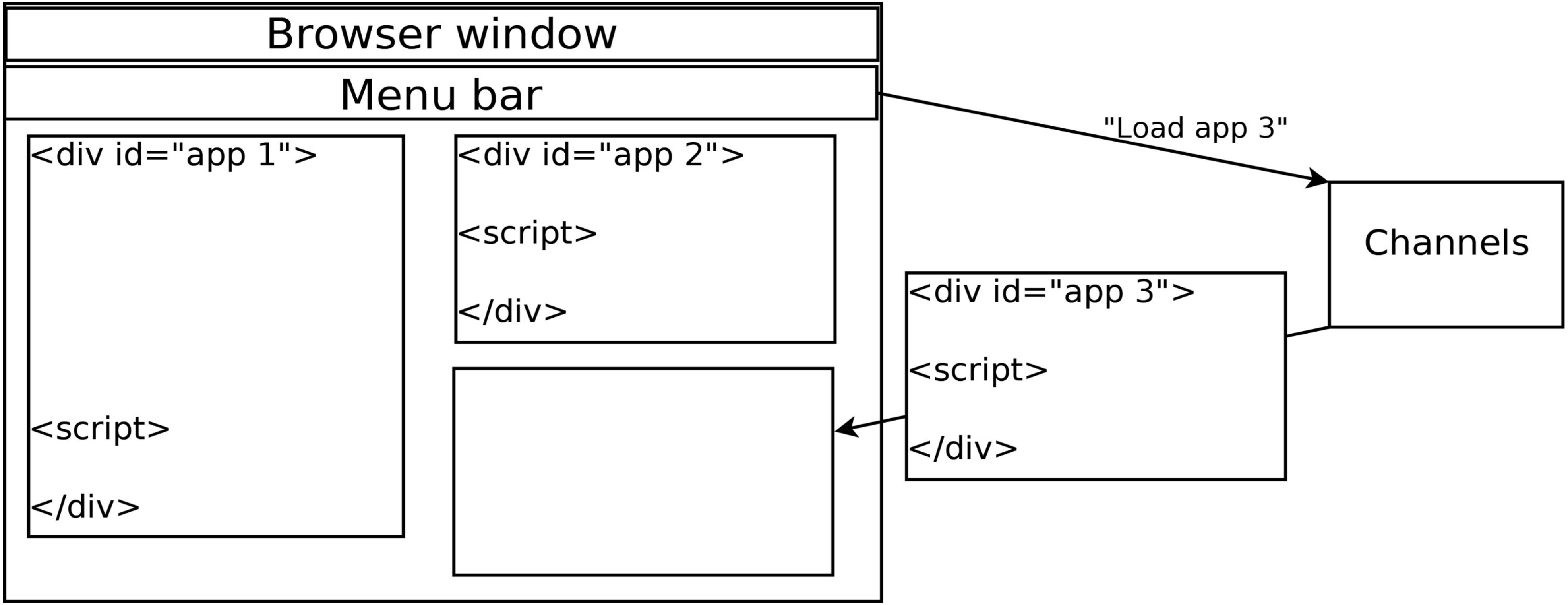
Coturn - https://github.com/coturn/coturn

- IP NAT traversal (Using `STUN` for address identification)
- Connection hub when direct connection isn’t possible (`TURN`)

The home page

- HTMX is loaded
  - Websocket extension is loaded
  - Custom transformResponse extension is loaded – static/js/tr.js
    - Assumes data coming through websocket is JSON
    - Looks for specific keys
      - “remove” – Removes an html element
      - “html” – HTML to be injected into the page as a normal htmx message
    - All other keys assumed to be for a “registered app”
    - Provides facilities to multiplex the websocket usage

Multiplexing a websocket



Websocket JSON - transformResponse

```
{
    ‘<app name 1>’: {
        ‘type’: ‘<event 1>’,
        ‘<key 1>’: ‘<data 1>’
    },
    ‘<app name 2>’: {
       ‘type’: ‘<event b>’
        ‘<key 1>’: ‘<data 1>’
    },
    ‘html’: ‘<div id=”new-div”>This is text being injected into the page</div>’
}
```

WebRTC Events

- RTC events received from consumer
  - connect – save the channel_name
  - other – open connection to one peer
  - others – open connections with multiple peers
  - signal – receive initial connection info from peers via Channels
  - disconnected – disconnect from peer

tr.js

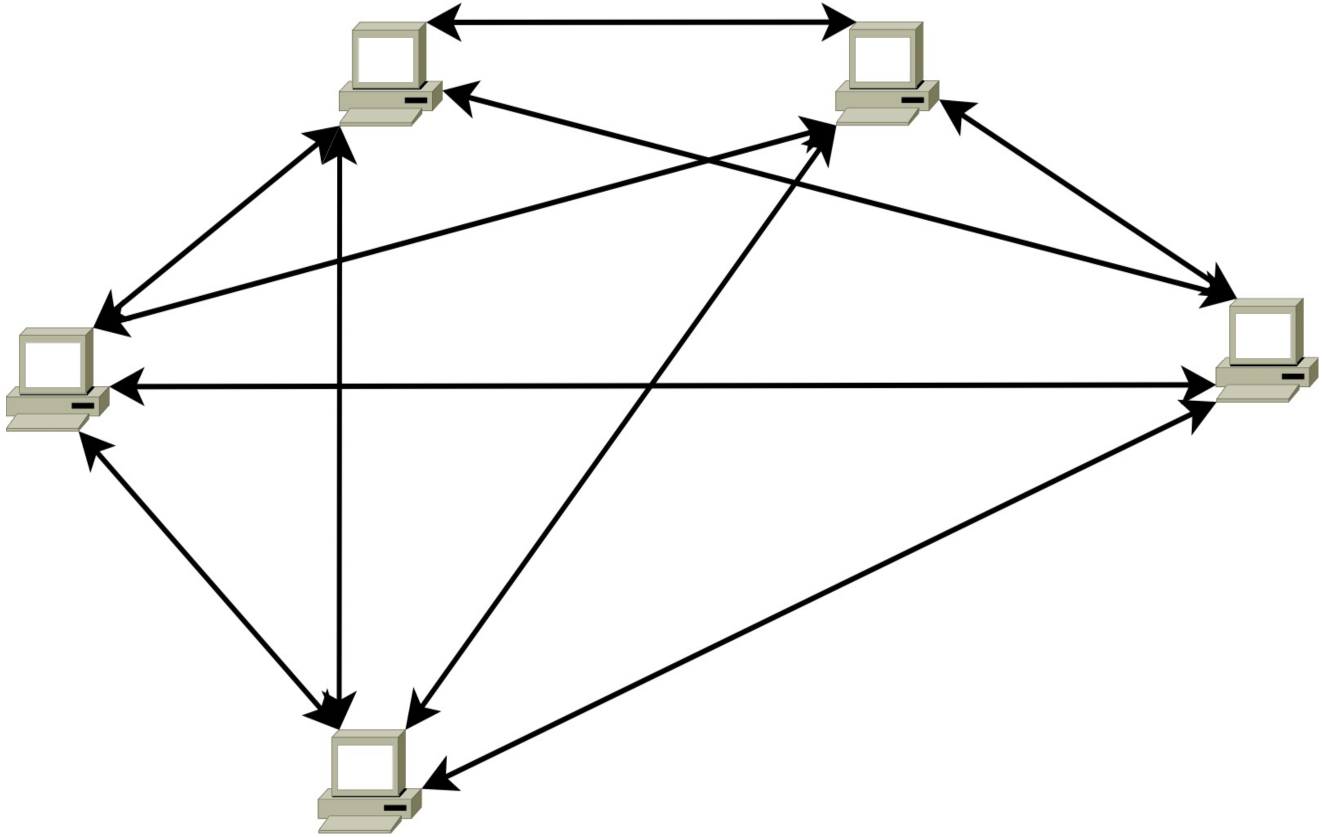
```
transformResponse : function(text, xhr, elt) {
    const data = JSON.parse(text);
    for ([app, message] of Object.entries(data)) {
            var event = message.type;
            apps._forward(app, event, message)
     }
```

```
'_forward': function(app, name, message) {
    if (apps[app] && apps[app].has(name)) {
        for (target of apps[app].get(name)) {
            target(message);
        }
    }
}
```

Client.js – Registering event handlers

```
apps._add('rtc', 'connect', connected);
apps._add('rtc', 'other', connected_other);
apps._add('rtc', 'others', connected_others);
apps._add('rtc', 'disconnected', disconnected_other);
apps._add('rtc', 'signal', signalled);
```

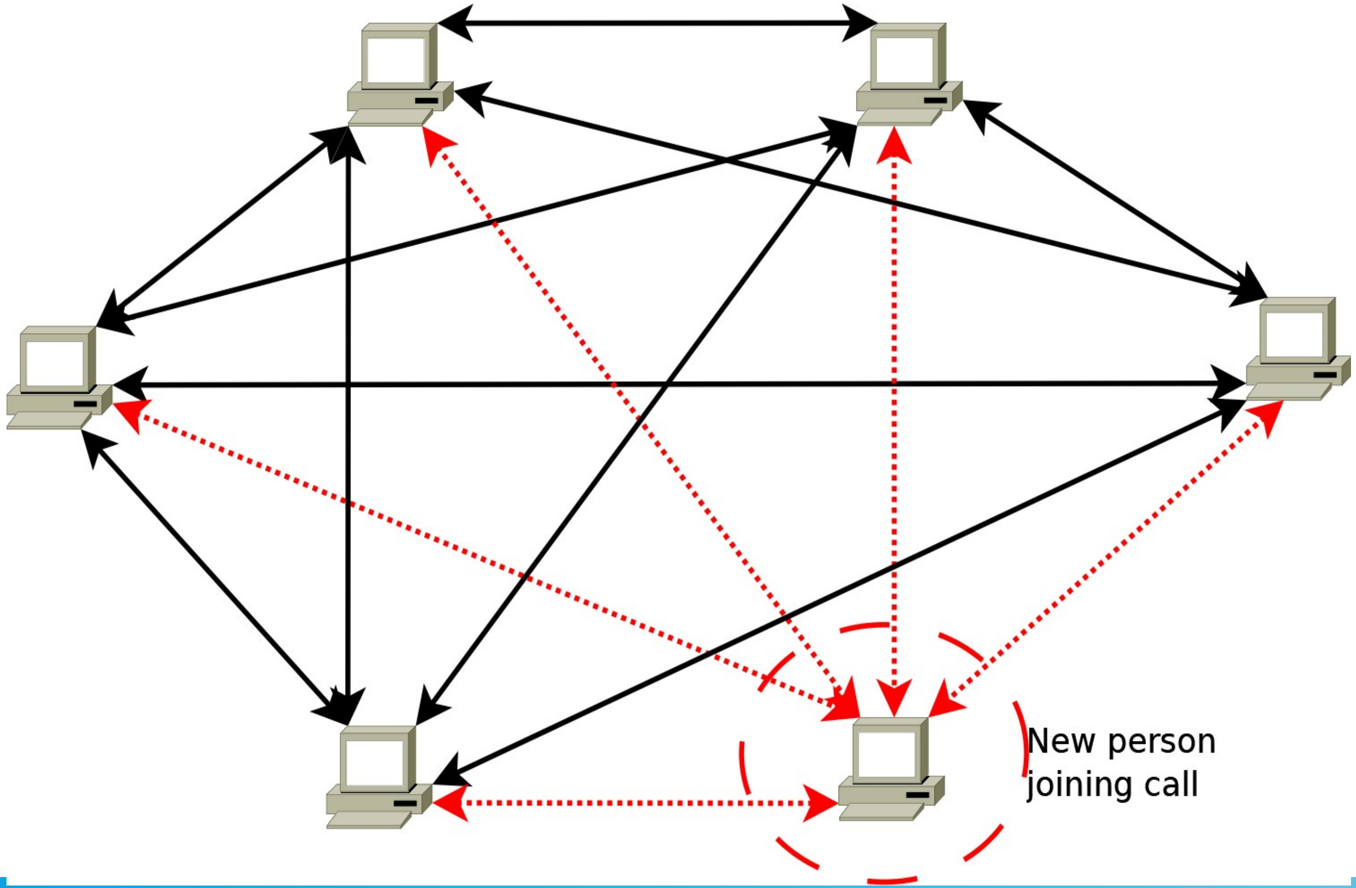
A group call in progress



Join Call – client.js

- Join Call → handleCallButton(event)
  - Sends message through websocket: {‘join’: ‘video’}

New person joining a group call



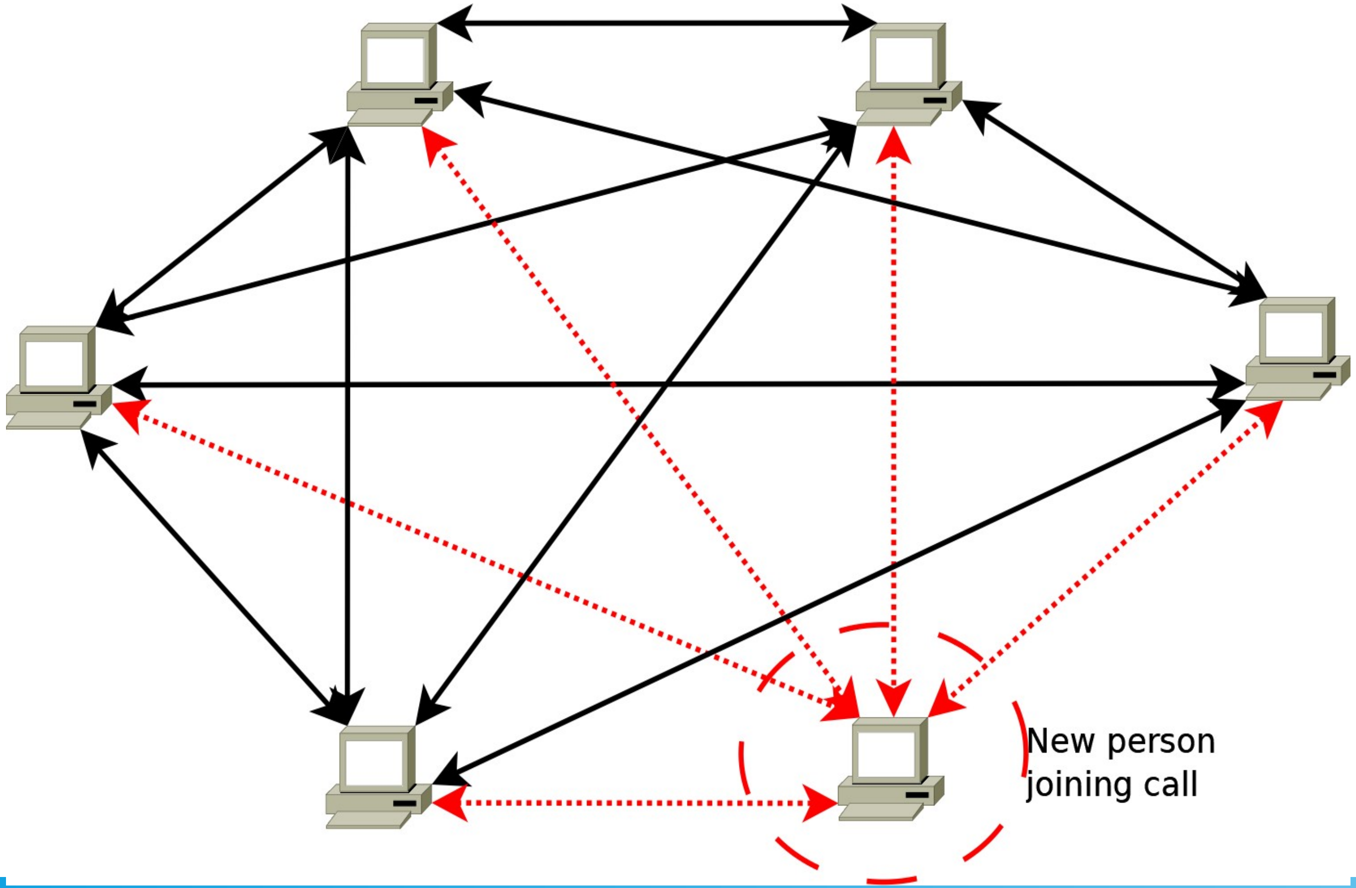
Join Call – In the Channels consumer

- Get a list of all other occupants in the room
- Create an html div for each other occupant, sends to self
- Create an html div for self, sends to all other occupants
- Send “other” signal to all other occupants in the room
- Send “Connect” event to self.

The WebRTC div

```
{% load static %}
<div id="others" hx-swap-oob="beforeend">
  <div id="{{id}}-div">
      <figure id="{{id}}">
        <video autoplay playsinline poster="{% static 'blank.png' %}">
        </video>
        <figcaption>{{user_name}}</figcaption>
      </figure>
  </div>
</div>
```

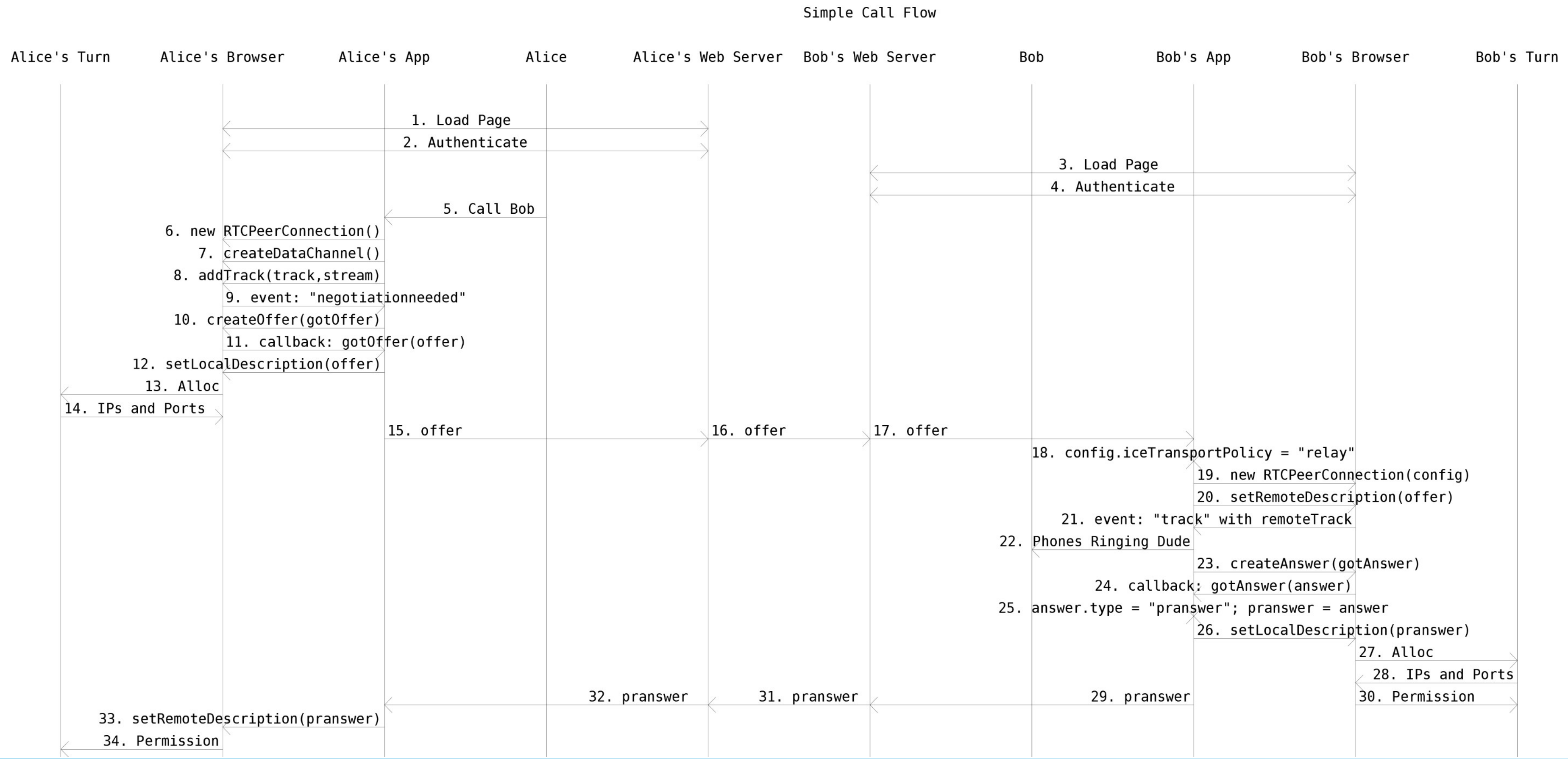
New person joining a group call



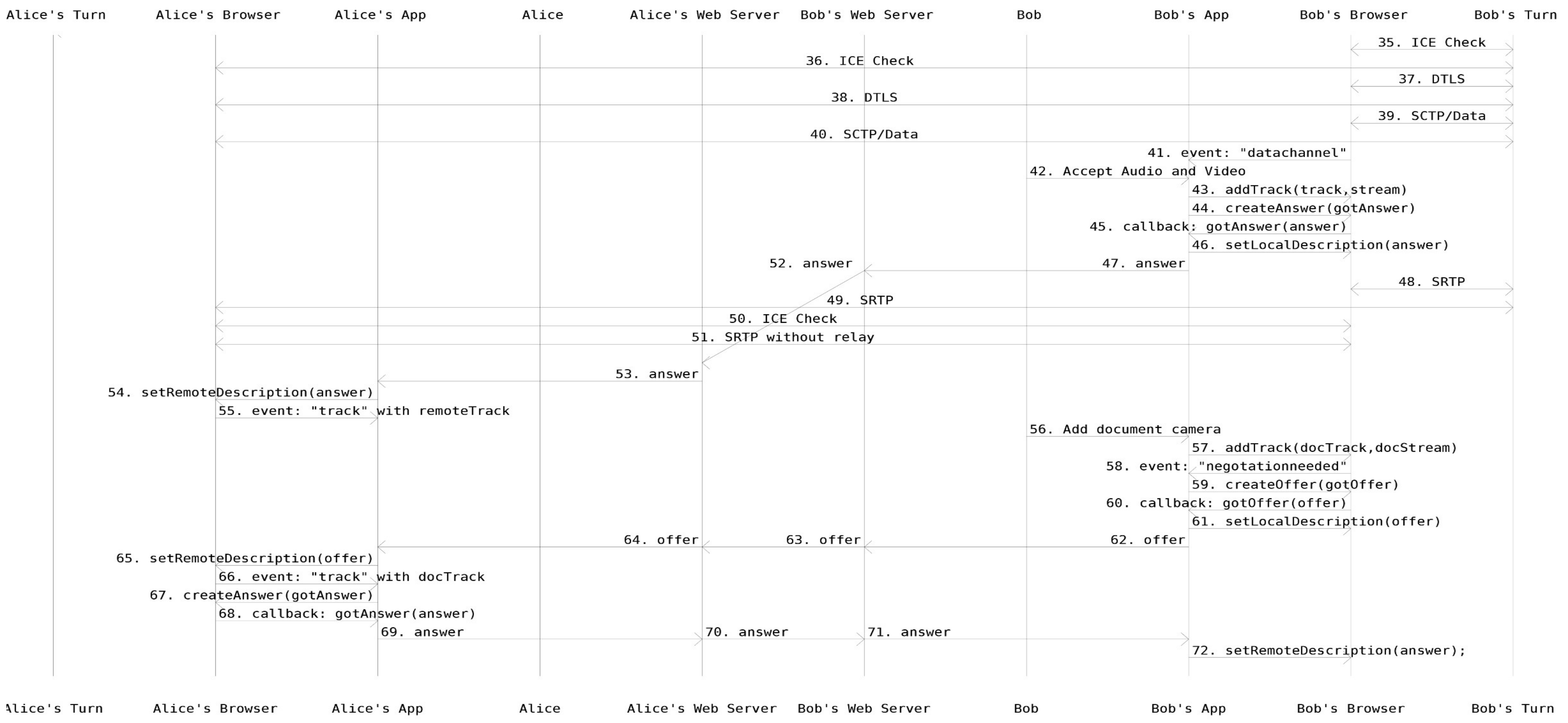
Negotiating a connection

- Both sides want to communicate with each other
- But, the real world is a messy place…

The “simplified” flow of events, browser-to-browser – pt 1



Part 2 – Complete docs at https://w3c.github.io/webrtc-pc/



What is negotiated?

- ICE – Interactive Connectivity Establishment
  - MDN defines ICE as:
    - A framework facilitating the connection of two peers, regardless of network topology.
  - Looks for the lowest-latency path:
    - Direct UDP
    - Direct TCP (through http or https, in that order)
    - Indirect, via TURN server

This is where coturn fits in.

- Open Source package, available for most distros
  - https://github.com/coturn/coturn

- Supports two key protocols
  - STUN: Session Traversal Utilities for NAT
  - TURN: Traversal Using Relays around NAT

Set your phasers for ...

- STUN: Session Transversal Utilities for NAT
  - Think of it as “What’s my IP” for WebRTC
  - Low traffic utilization
  - Many public servers available
  - Can configure multiple STUN servers for use
    - Greatly multiplies traffic as all paths are evaluated

When there’s no other path

- TURN: Traversal Using Relays around NAT
  - All data streams are passed through it
  - Extremely high traffic utilization
    - It can become a severe bottleneck in large group situations
  - You **NEED** to implement the security layer ...
    - … unless you have an unlimited budget for data
  - No public servers available …
    - … at least not for long

How are they defined?

```
rtc_config: {
    IceServers: [
        { urls: 'stun:kww.us:3478' },
        { urls: 'stun:stun.l.google.com:19302' },
        {
            urls: 'turn:kww.us:3478',
            username: "dcus",
            credential: "dcus2024",
        },
    ],
    iceTransportPolicy: "all"
},
```

What else is negotiated?

- SDP – Session Description Protocol
  - Describes the content of the connection
    - Resolution, Codecs, Encryption (if any), etc
  - Technically, a data format, not a protocol – sample:

    a=ice-ufrag:hyrq
    a=rtpmap:111 opus/48000/2
  - Can be 100 lines or more
  - Fortunately, you don't need to know **anything** about this. This is all handled by the browser. But…
    - These become messages through Channels
    - You **will** need to increase your channel capacity

Lots of data going back and forth

- There’s another aspect to this exchange …
- What happens when both sides initiate a connection at the same time?

Arranging a meeting between two people – the “Ideal”

- “Let’s meet in the lobby.”
-
- “How about 6 PM?”
-
- “Good. I’ll see you there”

-
- “Ok, what time?”
-
- “Great! I’ll see you in the lobby at 6 PM.”

What could happen

- “Let’s meet at the bar”
- “Ok, meet you in the lobby at 7 PM?”
- “Cool, I’ll be in the bar at 6.”
- (???)

- “Let’s meet in the lobby”
- “I’ll find you in the bar, 6 PM?”
- “Great, lobby at 7.”
- (???)

What *should* happen

- “Let’s meet at the bar”
- <pause>
- “Cool, I’ll be in the bar at 6.”

- “Let’s meet in the lobby”
- “I’ll find you in the bar, 6 PM?”
- “Great, see you then.”

The “Perfect Negotiation” protocol

- https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Perfect_negotiation
- Two roles are defined between the peers
  - Polite
  - Impolite
- The assignment is completely arbitrary
- The assignment must be deterministic
  - Everybody must understand the rules
  - In this case, the new caller is impolite

Once the negotiation is complete...

- The media streams are assigned to the “<video>” tag

```
<video autoplay playsinline poster="{% static 'img/placeholder.png' %}">
</video>
```

- Data Channels can also be created to share non-AV data
  - Static images
  - Text

Things not covered here

- Audio
- Data channels
- Multiple rooms
- Coturn security
- Room cleanup
- Multiple turn servers

- Diagnostics and troubleshooting
  - chrome://webrtc-internals/
  - Firefox developer tools

- **Deployment**
  - See the notes in README.MD

Thank you!

Best place to find me?

https://forum.djangoproject.com/

(And I’m here tomorrow and Friday morning)