WebRTC

How we've been doing lectures
this last few weeks...

Various bits of this follow:
https://github.com/webrtc-security/webrtc-security.github.io/blob/master/index.md

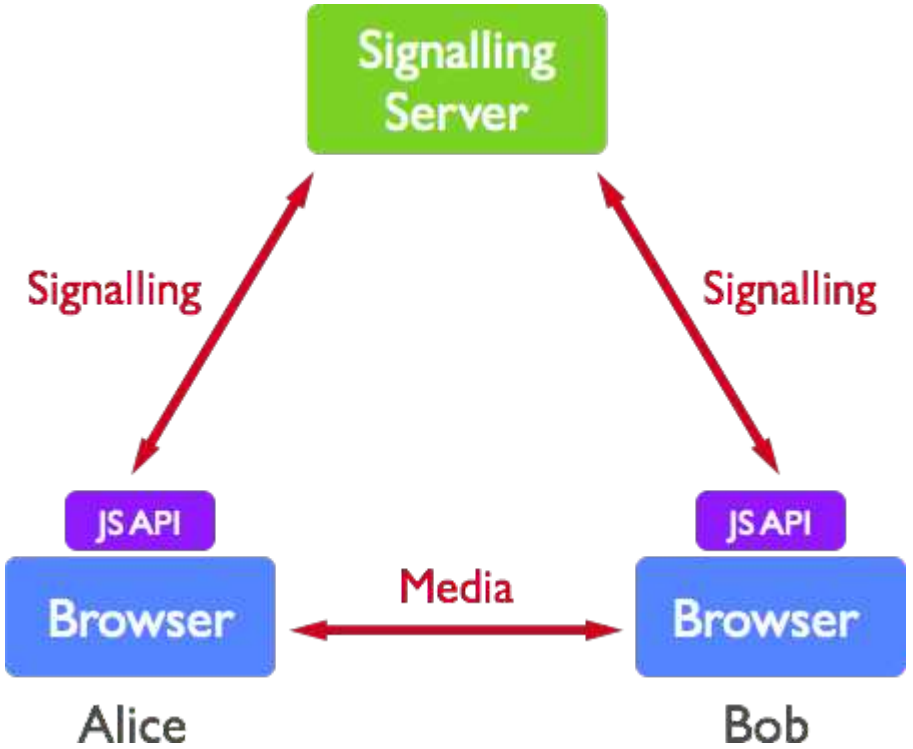
WebRTC Functions

- WebRTC is a way to make encrypted voice/video calls **directly** between two web browsers
  - Ideal flow is direct ciphertext packets from one browser to the other and that works just fine e.g. with IPv6 at home, that is; in a **peer-to-peer** mode
- A web site mediates call setup using Javascript and standard interfaces implemented in web browsers, and calls use standard protocols for data, voice and video
  - This sort of means that the signalling protocol is non-standard (up to whatever the JS developer wants) but the media path uses standard protocols (v. different from POTS!)
- Uses: sales and support for web sites, conference calling for enterprises, calling between random people

WebRTC Standardisation

- Co-operation between IETF (protocols) and W3C (browser APIs), largely driven by browser-makers but with lots of input from more traditional telcos
  - https://datatracker.ietf.org/wg/rtcweb/
  - https://www.w3.org/2018/07/webrtc-charter.html
- Mostly smooth but took way longer than expected due to inherent complexity and a major bun-fight (>2 years) about which audio/video codecs to make mandatory-to-implement
  - Such codecs have historically been patented – the telco industry was fine with that but with WebRTC open-source browsers replace handsets so patents and odd IPR licenses become much more of an issue
- Standardisation-nerd lesson: don't create a huge web of interdependencies in specifications – approx 100 almost-finished RFCs have been queued up waiting on one another for up to 5 years!
  - https://www.rfc-editor.org/cluster_info.php?cid=C238

WebRTC Basic Flow



WebRTC – the bad side

- Web site that sets up call can eavesdrop, maybe beyond life of call
  - Can be detected in many cases, so less likely to be done commonly, but there are many bad actors in the world
- “Consent” model is as broken as ever, but WebRTC extends SOP security model and “consent” model (for camera/mic) so is therefore making a bad situation worse
- Peer-to-peer connections (a good thing) require exposing information about the peers’ IP addresses to work – sometimes those addresses can be sensitive
  - OTOH WebRTC probably exposes many new correlators/identifiers despite the designers’ best intentions
  - And they have tried to eliminate new privacy leaks where possible

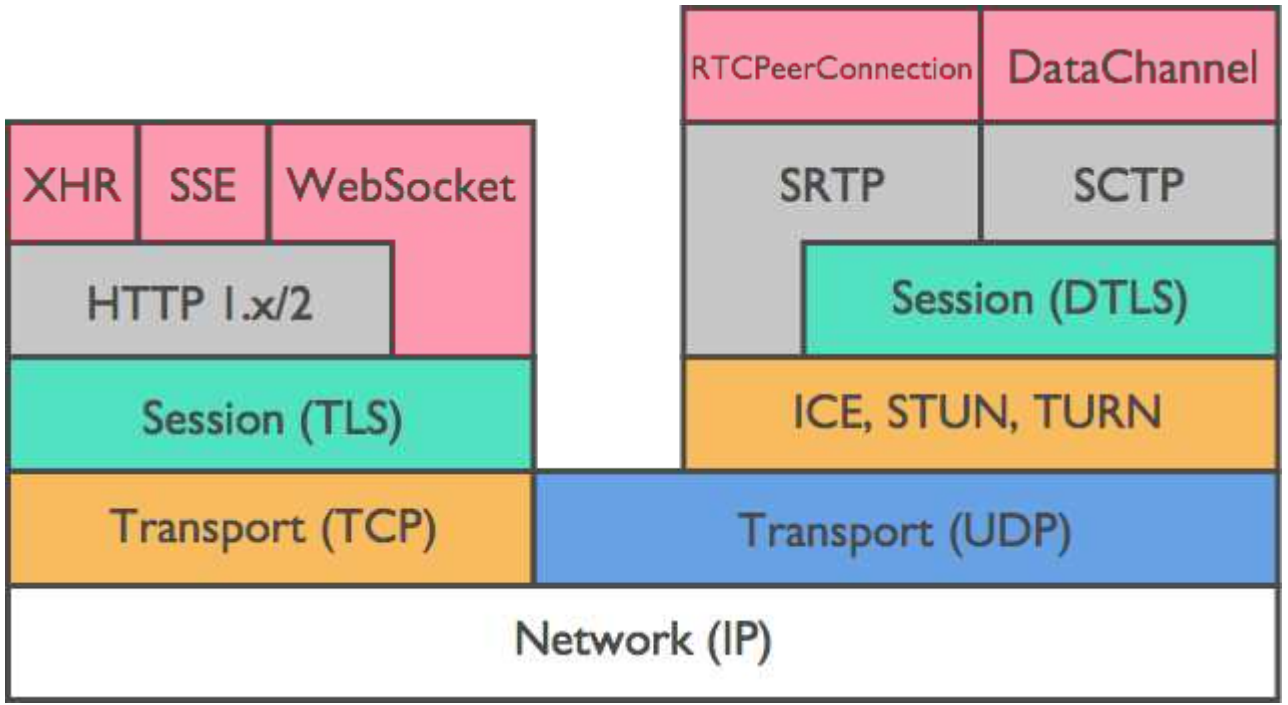
WebRTC – the good side!

- You can be your own phone company, https://jell.ie/webrtc/ is mine
  - That's all built from open-source, I didn't write any WebRTC code at all, all I did was download, build and configure stuff, and I didn't have to ask anyone for permission (figuring out how is time consuming but doable)
- https://meet.jit.si/somerandomname is a slightly more scaleable WebRTC deployment and various others exists at that scale
- Highly scalable tools like Cisco's Webex use WebRTC without needing special client s/w; I've been on "calls" with >200 people that replaced a cancelled f2f conference session
- Blackboard's "collaborate ultra" feature is also a WebRTC instance
- In COVID19 lockdowns, WebRTC has allowed many, many people to continue to work from home without having to install weird conference calling clients (not all video calling tools use WebRTC)

WebRTC Protocol Stack

- The WebRTC protocol stack is incredibly baroque and complex
  - All the usual HTTPS stuff, plus...
  - Weird mixture of DTLS, SCTP, UDP, SRTP
  - With added ICE, STUN and TURN for NAT fun
  - And SDP for offer/answer fun
- Reasons for WebRTC complexity:
  - Browsers don’t normally talk to one another
  - TCP isn’t great for real-time media
  - Negotiating media parameters (codecs, bit rates, screen details) is complex
  - Complicated corner-case use-cases: legacy interop, conference calls

WebRTC Stack



WebRTC API Terms

- XHR – XMLHttpRequest is a (now outdated by fetch()?) way to write asynchronous JS code
- SSE – Server Sent Events API (uncommon?)
- WebSockets – provide a way to send arbitrary data from a browser to a peer
- RTCPeerConnection – is a new WebRTC API that allows a browser tx/rx real-time data with a peer browser in a WebRTC session
- DataChannel is another WebRTC API, but for non-realtime data, e.g. the chat session in a call

SDP for media

- Session Description Protocol (SDP, RFC 4566) was designed before WebRTC for VoIP
- Offer/answer model where one peer says e.g. “I can do AV8, screen: 1024x768” etc. and other peer answers – kind of like TLS ciphersuite negotiation but with many more degrees of freedom
  - Developers generally hate SDP as it’s exposed to the JS coder

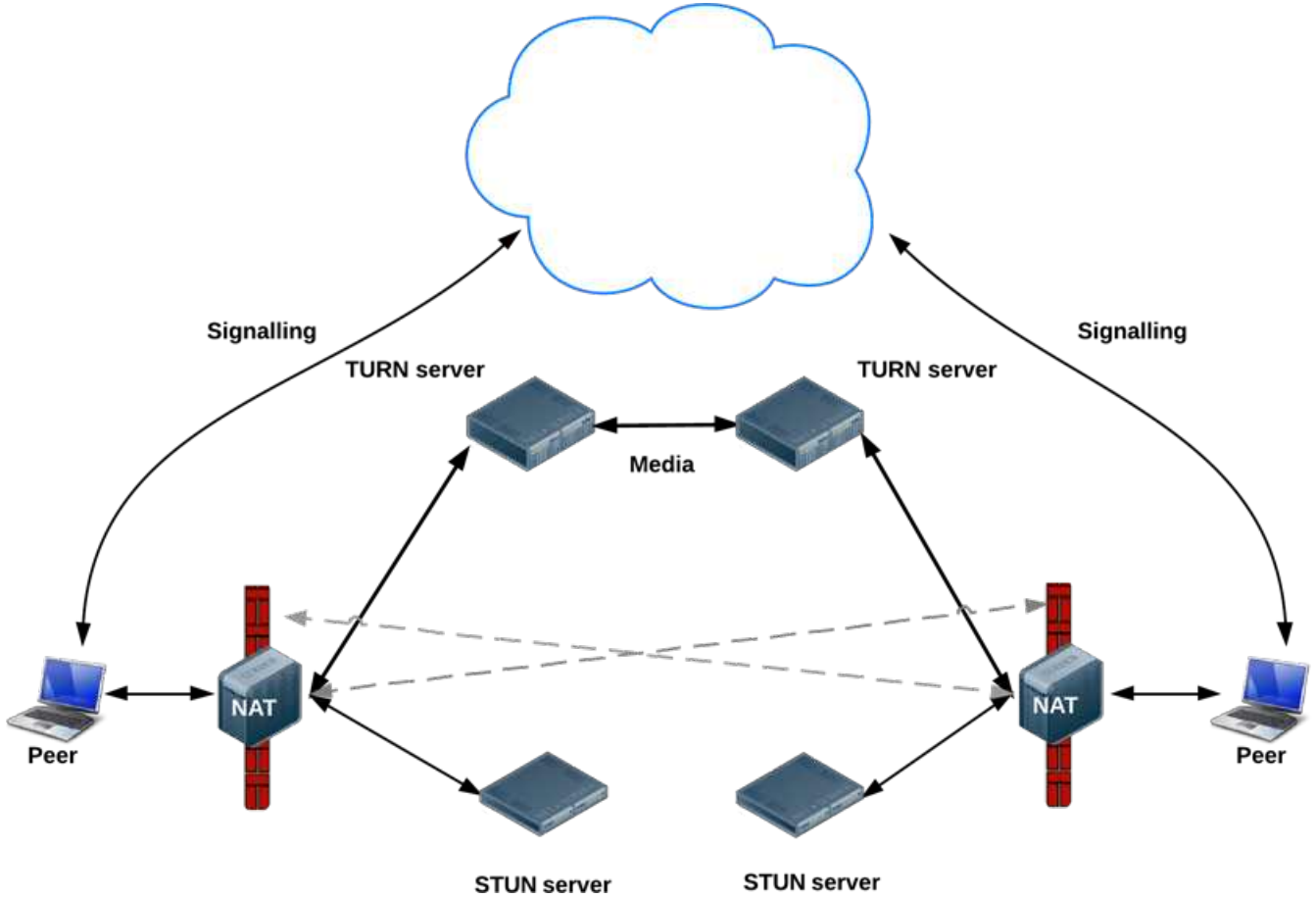
WebRTC Protocol Terms

- SRTP – is the Secure Real Time Protocol, RFC 3711 defines a way to secure real-time media traffic
- DTLS – is the Datagram Transport Layer Security protocol, RFC 6347
- DTLS-SRTP, RFC 5764 – defines how to use DTLS session keys for SRTP
- SCTP – is the Stream Control Transmission Protocol, RFC 4960 – basically provides a mix of TCP-like and UDP-like features

WebRTC and NATs

- Many browser instances will be behind a NAT
- ICE – Interactive Connectivity Establishment (RFC 8445) is an algorithm for picking/agreeing whether to communicate directly or not, and if not, how – may use STUN and/or TURN
  - Also handles IPv4 and IPv6 and multiple interfaces (e.g. WiFi and GSM)
- STUN – Session Traversal Utilities for NAT (RFC 5389) – a STUN server on the public Internet enables a browser to figure out it's public IP address
- TURN – Traversal Using Relays around NAT (RFC 5766) provides a way to relay packets if there's no other way to get them inside the NAT
- Mostly: TURN servers are also STUN servers, but STUN servers are cheap (no media) whereas TURN servers can be expensive (media traffic goes via TURN server), so many STUN servers might not (willing to) offer TURN service
- STUN and TURN servers are capable of collecting call meta-data

WebRTC and NAT



From https://www.html5rocks.com/en/tutorials/webrtc/infrastructure/

IP Location Privacy

- ICE tries all the options (local addresses) it can to find the best path, ideally direct browser to browser without TURN
- If a user is using a VPN, that can be considered sensitive – ICE might publish that fact to the network, and the network is sometimes the adversary
- Took a couple of years to find a fix, eventually FF did
- Problem was raised by privacy advocates who work in places with repressive regimes
- Technical solution multicast DNS! (mDNS)
  - For any sensitive addresses, invent a “<random>.local” DNS name
  - If two browser both in same scope (e.g. on same LAN) then could resolve mDNS name
  - If not, then sensitive address not exposed!
  - Loosely based on how Apple do discovery in LANs (bonjour)
  - https://tools.ietf.org/html/draft-ietf-rtcweb-mdns-ice-candidates

WebRTC Security Overview

- Documented in:
  - https://tools.ietf.org/html/draft-ietf-rtcweb-security

Permission Model

- Browser Same Origin Policy (SOP) extended – user prompted to allow access for **site** to **microphone/camera** and that permission is **sticky** if the site uses HTTPS
  - When WebRTC started, HTTPS was not as ubiquitous as today, likely plaintext HTTP will be blocked or already is
- Use of microphone/camera likely indicated via icon on browser URL bar, and that's also how a user can **revoke permission** (if they remember – would they?)
- People on phones/tablets have less control (as always)
- Overall effect: web sites get permission is my bet – not sure if anyone has tried to measure how many people say "no" (and keep saying "no")

Screen Sharing

- Conference calling typically requires screen (or application window) sharing of some form for the “presenter”
- Handled via same permission model
- That’s a very powerful permission to grant any web site, and is hard for browser to implement well
  - What else is on your screen?

Transport Security

- DTLS used for DataChannel
- DTLS-SRTP used for media
  - DTLS-SRTP uses the (D)TLS handshake and then exports sessions keys for use with SRTP and is run between browsers (after ICE is done)
  - So web site doesn’t see the keys (but it could cheat if it wanted and add itself to a calls)
- But… this is between browsers neither of which has a public key certificate, so how’s that gonna work?
  - Unauthenticated, mostly – exchange digests of public keys in signalling and then treat those public keys as trusted (for that call) in TLS handshake
  - Theoretically, one can link to an identity provider (IdP) but not clear how common or realistic that may be
  - One use-case can provide authentication: if a person clicks on the “talk to an agent” button on a web-site then the web-site end of the call will be a server in a call centre that can have a good TLS server certificate

Browser debug info

- Browsers have debug interfaces for WebRTC:
  - Firefox: about:webrtc
  - Chromium: chrome://webrtc-internals
  - Opera: opera://webrtc-internals
- You can see what ICE did and the full horror of SDP:-)

Conference calls

- WebRTC can be used for conference calling
- Conference calling always has issues with how to do key management and allow e.g. late joiners to the call
  - That's a potentially and actually very tricky problem – see https://tools.ietf.org/html/draft-ietf-mls-architecture
- Almost never peer-to-peer flows because a server is needed for mapping e.g. video to clients' screen-size/codec etc.
- Zoom doesn't do WebRTC but something similar but proprietary
  - https://theintercept.com/2020/04/03/zooms-encryption-is-not-suited-for-secrets-and-has-surprising-links-to-china-researchers-discover/
- Jitsi.org's "meet" service does WebRTC though
- So does Cisco's Webex
- And Blackboard, which is what we're using now

Chrome://webrtc-internals for BB

https://eu.bbcollab.com/collab/ui/session/join/5a73a39528dc4126adf750f906f0ea51, { iceServers: [turn:ultra-eu-prod-turn.bbcollab.cloud:50000?transport=udp, turn:ultra-eu-prod-turn-tcp.bbcollab.cloud:443?transport=tcp, turns:ultra-eu-prod-turn-tcp.bbcollab.cloud:443?transport=tcp], iceTransportPolicy: all, bundlePolicy: max-bundle, rtcpMuxPolicy: require, iceCandidatePoolSize: 1, sdpSemantics: "plan-b" }, {advanced: [{enableDtlsSrtp: {exact: true}}, {enableDscp: {exact: true}}]}

Try it and see

- You can setup your own concall with no account at: https://meet.jit.si/
  - CAUTION: you may encounter anything at a fairly well-known URL!
  - The pathname in the URL can be whatever you want, a couple of other services do the same kind of thing
- I like Jitsi for calls with maybe up to 6 people
- I generally see problems with Jisti if the mix of network configurations gets complex, which it inevitably does with more participants
- Jitsi code:
  - WebRTC meetings code: https://github.com/jitsi/jitsi-meet
  - Videobridge: https://github.com/jitsi/jitsi-videobridge
- Jitsi people read the news too:-) https://jitsi.org/news/security/