

# Open Screen Protocol

## Day 1

---

Peter Thatcher ([pthatcher@google.com](mailto:pthatcher@google.com))

Mark Foltz ([mfoltz@google.com](mailto:mfoltz@google.com))

TPAC 2018

# Day 1 - Outline

Open Screen Protocol Overview

Specifics for mDNS/DNS-SD

Specifics of QUIC

Specifics of CBOR

Specifics for how to do JPAKE

Presentation API messages

Remote Playback API messages

# Second Screen CG History

- Nov 2013: Initial charter
- Nov 2013 - Dec 2014: Incubation of Presentation API
- Dec 2014: Presentation API transitioned to Second Screen Working Group
- Sept 2016: CG rechartered to focus on interoperability
- 2016-2017: Requirements, protocol alternatives, benchmarking plan
- Sept 2017: F2F at TPAC
- Jan-Feb 2018: SSWG rechartered. Phone conference, work plan
- May 2018: Berlin F2F
- October 2018: Here we are :-P

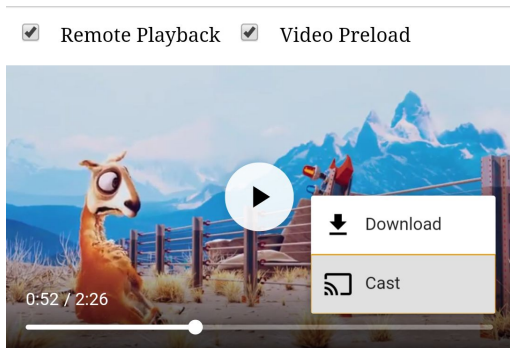
# Presentation API

1. **Controlling** page (in a browser) requests presentation of a URL on a **receiver** device (on a connected display).
2. Browser lists displays compatible with the URL; the user selects one to start the presentation.
3. Controlling and receiving pages each receive a **presentation connection**.
4. The connection can be used to exchange messages between the two pages.
5. Either side may close the connection or terminate the presentation.

# Remote Playback API

`<audio>` or `<video>` element can:

1. Watch for available remote displays
2. Request remote playback by calling `video.remote.prompt()`
3. Media commands are forwarded to the remote playback device



# Community Group Rechartering & Scope

Address main feedback from TAG around **interoperability**

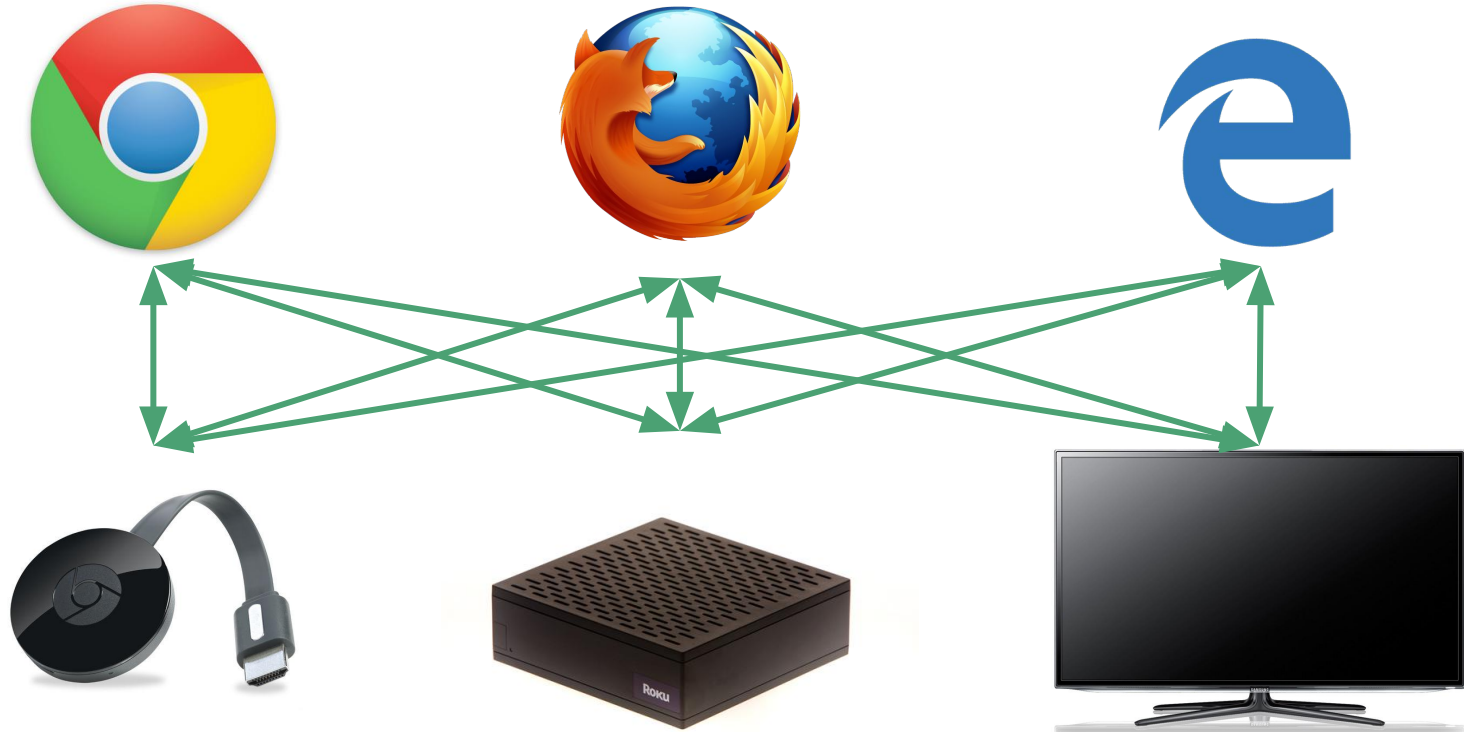
Controllers and receivers on **same LAN**

Presentation API: 2-UA mode (**“flinging” URL**)

Remote Playback API: Remote playback via a **src=“...” URL**

**Extension** ability for future use cases

# Open Screen Protocol



# Open Screen Protocol

Specify all network services & protocols needed to implement APIs

Can be deployed across a variety of devices and platforms

Re-use modern protocols and cryptography



# Functional Requirements

1. Discovery of presentation receivers and controllers on a shared LAN
2. Implement Presentation API
  - a. Determine display compatibility with a URL
  - b. Creating a presentation and connection given a URL
  - c. Reconnecting to a presentation
  - d. Closing a connection
  - e. Terminating a presentation
3. Reliable, in-order message exchange
4. Authentication and confidentiality
5. Implement Remote Playback API for `<audio>` and `<video>` `src=`

# Non-functional Aspects

Usability

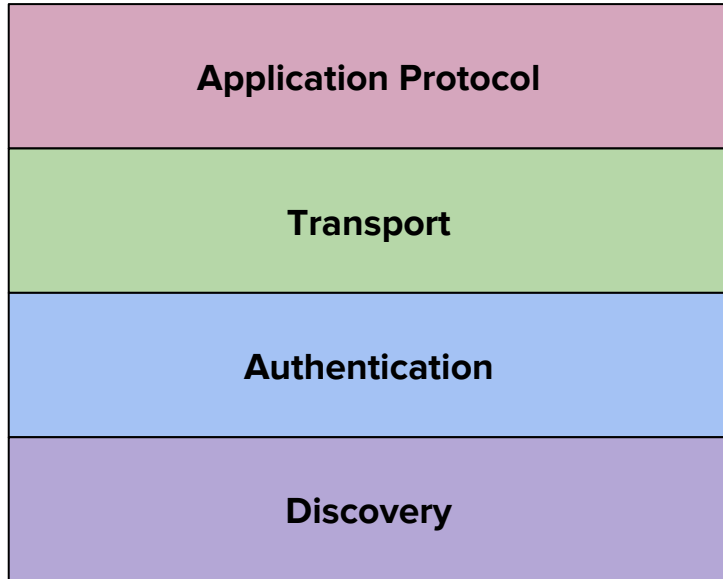
Preserving privacy and security

Resource efficiency (battery, memory)

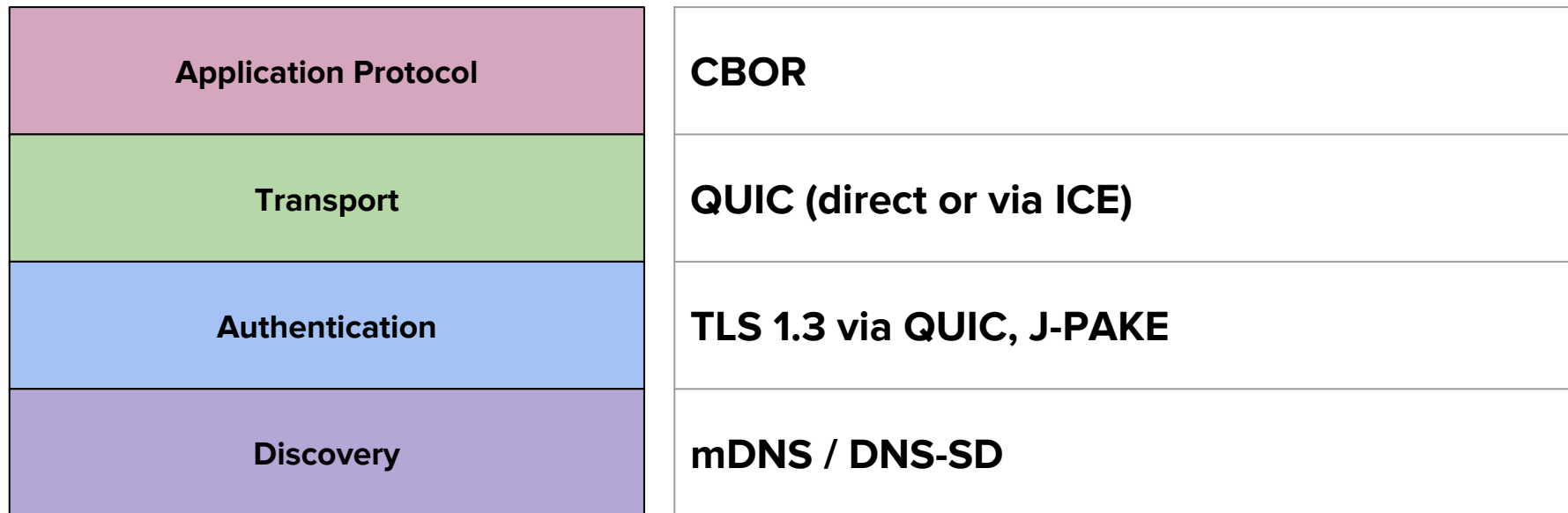
Implementation complexity on constrained devices

Extensibility and upgradeability

# Open Screen Protocol - Stack



# Open Screen Protocol - Current Approach

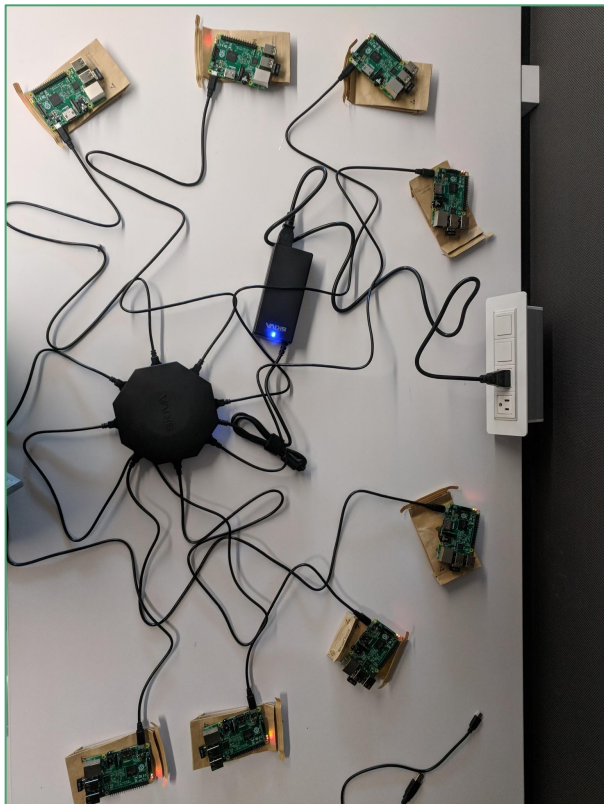


# Evaluation & Benchmarking

For each technology/protocol,

- Write up proposal for how it could be used
- Evaluate against requirements (performance, security, implementation)
- Write up proposal for benchmarking performance in the lab

# "Open Screen Lab"



# What has been accomplished

- Requirements analysis, research into alternatives
- Consensus to specify "modern" protocol stack at Berlin F2F
  - Discovery, transport, protocol format
- Authentication approaches proposed
- Open source implementation announced & in progress
- V1 spec document started and in progress
- Benchmarking plan proposed

# Major work items remaining for “V1”

- Finalize details on mDNS usage
- QUIC connection handling
- CBOR based control protocol
- Remote Playback API support
- Specify authentication mechanisms
  - Integrate J-PAKE
  - Support public-key based authentication with TLS
- V1 spec document written :-)
  - And reviewed...
- Benchmarking, data collection



# Forward-looking work

- **Media streaming**
- **LAN traversal (ICE) support**
- Backup/alternate discovery
- WebRTC / ORTC polyfill implementations
- Other "V2" features

# Things to talk about

- mDNS/DNS-SD
- QUIC
- CBOR
- Protocol messages with CBOR, defined in CDDL
- JPAKE

# mDNS/DNS-SD

Some details to iron out:

- Service name
- TXT metadata





# Service Name

RFC 6763 Section 7 basically says it must be **`_X._udp.local`** with more rules for `x`.

How about **`_openscreen._udp.local`**?

FYI, this needs to be registered with IANA.



# TXT Info

We can put **string key/value pairs** in the **mDNS** discovery.

We also have some **device info we'd like** to convey.

Two options:

A: Put device info in TXT info

B: Minimal things in TXT info; device info in CBOR over QUIC



# Option A: info in TXT

- Protocol Version (**ve = 1**)
  - (for major protocol changes like not using QUIC or CBOR or JPAKE)
- Device ID (**id = GUID**)
- Device Friendly Name (**fn = Living Room TV**)
- Device Model Name (**mn = LG 55UJ6300**)
- Device Type? To show an icon
- Device capabilities (**ca = bitfield**)
  - receive audio, receive video, send audio, send video

Pro: We know the info before doing the QUIC handshake

**Con: No authentication/encryption of this metadata**



## Option B: bare minimum; rest over QUIC

- Protocol Version (**ve = 1**)
- Device Fingerprint (**fp = XXX**) and Timestamp (**ts = timestamp**)  
(To know if you should connect to get new metadata or not)

```
device-info-request = {  
  request  
}  
  
device-info = {  
  0: text ; friendly-name  
  1: text ; model-name  
  ; ...  
}
```

```
device-info-response = {  
  response  
  1: device-info;  
}
```

**Pro: now encrypted/authenticated**

Con: 1 more RTT when connecting to new device

Note: may need to share this the first time before authentication



# QUIC

Some things to iron out:

- How many QUIC connections?
  - If a QUIC server can accept many QUIC connections and demux them
  - If we need more than one for more than one tab/profile
- When/how should QUIC connections be kept alive?
- How messages map to QUIC streams





# How many QUIC connections

**One QUIC connection per (client, server) pair**

There isn't a reason to make more than one.

However, if a client does create more than one, they can be demuxed by connection ID as long as the QUIC packets include the connection ID (they can be omitted if there is only one connection per 5-tuple)



# What about more than one tab/profile?

A browser can serve more than one tab/profile using one QUIC connection, but **may choose to use different network QUIC connections (and ports)**

Incoming messages of presentation connection messages, remote playback messages, etc, can be routed to the correct tab/profile based on IDs in messages.

The fact there is one QUIC connection underneath (or many) is an implementation detail that doesn't affect the APIs or web apps. This is because none of the QUIC Stream IDs are meaningful.

However, that may allow the remote endpoint to correlate profiles, in which case a browser may choose to use separate QUIC connections (and ports) from different profiles.



# Can a server demux more than one client?

**Yes.**

If clients have different (ip, port), then the server can demux on that.

If a client uses the same (ip, port) for more than one connection, connection ID can be used to demux.



# When QUIC connections should be kept alive

Keeping connections alive allows the client to receive messages from the server.

But keeping connections alive may also use battery.

**Recommendation: Keep a QUIC connection alive if you're a client that needs to receive messages (such as presentation connection messages) or a server that needs to send them. Otherwise, close the QUIC connection and reconnect when you need to (treat QUIC connections as ephemeral)**

Also allow apps to keep connections alive if they have per-connection costs (auth)

TODO: Add an API for controlling connection timeout (default: none)

Maybe considering allowing clients to give servers a port they can reconnect to if desired.

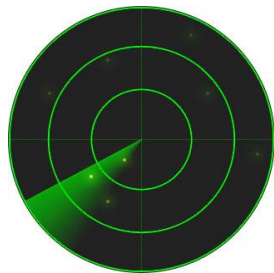


# How QUIC connections should be kept alive

If you're on a LAN, you technically could keep the QUIC connection state in memory indefinitely. If the other side just disappears (unplugged; battery dies), you wouldn't know.

So, **it's nice to ping the remote side occasionally** to make sure it's still there.

But how and how often?



# How to send pings

- **ICE**

- ICE already does regular pings, so if/when ICE is used with OSP, we won't want or need extra pings.
- But using ICE for on-LAN cases a lot of extra complexity for something very simple.

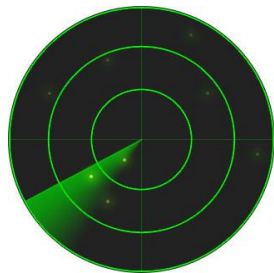
- **QUIC**

- QUIC has built-in ping/pong frames.
- But QUIC libraries won't necessarily expose the ability to send or receive them to apps
  - (Chromium's impl allows sending easily, but not receiving)

- **OSP**

- We could send pings in QUIC stream as part of OSP
- It's more to define, but we can also piggyback status info
- Works even if we use something other than QUIC as the substrate someday

**Recommendation: add a ping/status message to OSP**



# How often to send pings

More often: uses more network/battery

Less often: learn later that a device has gone away

ICE magic numbers are every 20-30 seconds

## Recommendation:

- **Send every 25s** (unless other request/response traffic fulfills same need)
- **Unless you're using ICE**; then only when you need to send or recv fresh info



# How to map messages to QUIC streams

QUIC streams are **indefinitely long**. Put in as many messages as you want.

Messages in a QUIC stream are **ordered**.

Messages in separate QUIC streams are **not ordered** relative to each other.

If you want messages out of order, use **separate QUIC streams**.

If you want messages in order, use **one QUIC stream**.

**Recommendation: 1 group of ordered messages = 1 unidirectional QUIC stream**

**Recommendation 2: Ephemeral QUIC stream IDs (but reserve streams 1-10 for future)**



# CBOR

- How do we put multiple CBOR messages in a QUIC stream?
- How do we know the type of a CBOR message when it arrives?
- What kind of “keys” do we use?
- How do we encode time stamps/durations?
- Use of CDDL

# Putting multiple CBOR messages in a QUIC stream

Length-prefixed messages can always work, but it uses up some bits per message.

It looks like CBOR doesn't require it unless we use size = 0 ("rest of stream").

We could just not do that and put CBOR messages "back to back".

**Recommendation 1: don't use "rest of stream"**

**Recommendation 2: CBOR messages "back to back";**

Consider adding length-prefixing if it will make parsing easier (a parsing library might not parse streams as well as fixed-size buffers).

# Tagging CBOR messages

When reading the bytes coming in the QUIC stream, you need to tag the type of message some how.

**Option A:** Use CBOR's **built-in tagging**. Easy! But you're kinda supposed to register with [IANA](#) to avoid collisions (easy for us, not so much for extensions).

**Option B:** Treat the QUIC stream as a CBOR **array of (type, value)**. A little more clumsy and inefficient, but avoids ID collision/IANA issue.

**Option C:** Use **type-prefixing** in the QUIC stream. More efficient than either and doesn't require IANA registry. But can't define in CDDL.

**Recommendation: Use type-prefixing (Option C); maybe put a comment in CDDL**

# Intro to CDDL

```
; This is a comment
person = {
  ; The key is a string; the value is a uint
  age: uint
  ; The key is an int; the value is a string; the whole thing is optional
  ? 1: string ; username
  ; The key is an string; the value is an array of strings
  favorite-foods: [* string]
}
```

```
; This is an enum; notice that “,” is optional above
color = &(amp;blue=1, red=2, green=3)
```

```
; The “keys” are positions/index
coordinate = [x: uint, y: uint]
```

# Tagging CBOR values (“key” type)

In CBOR, you can put values in an message/object/group in 3 ways:

**Positional** (like an array): Most efficient, least flexible

**String keys** (like JSON): Least efficient, most flexible

**Int keys**: Rather efficient, rather flexible, less readable in CDDL and on the wire

**Recommendation: Use int keys** with comments for the field name

Consider positional keys (arrays) for certain situations (streaming, tuples)

# Encoding timestamps/durations

There are a few ways we can encode timestamps:

**CBOR-defined timestamps:** builtin, but can be encoded as floats

**uint microseconds:** not builtin, but just as easy, really, and we have more control over the encoding

**Recommendation: use microseconds unless there's a reason to use a different timescale (such as with audio and video)**

# Protocol messages defined in CDDL

Messages for different purposes:

- Base messages/types (used by all the rest)
- Connection-level things (not tied to any API)
- Presentation API
- Remote Playback API
- Streaming

# Base messages/types

- Request/response
- Common error codes and types



# Request/response

```
; embedded/included/subclassed in each request  
request = (  
  0: request-id ; request-id  
)
```

```
request-id = uint
```

```
response = (  
  0: request-id ; request-id  
)
```

# Common error codes and types

```
result = (  
  success: 1  
  url-not-valid: 10  
  invalid-presentation-id: 11  
  timeout: 100  
  transient-error: 101  
  permanent-error: 102  
  unknown-error: 199  
)
```

```
microseconds = uint  
microseconds-range = [  
  start: microseconds  
  end: microseconds  
]
```

# Connection-level things

- ping/pong/status
- maybe JPAKE handshake
- maybe ICE candidates

# Ping/pong/status

```
status-request = {  
  request  
  ? 1: status ; status  
}
```

```
status-response = {  
  response  
  ? 1: status ; status  
}
```

```
status = {  
  ; could be something like this:  
  ; 0: microseconds ; system-clock  
}
```

# JPAKE

Coming tomorrow :)

# Presentation API messages

- URL availability
- Initiate/terminate presentation
- Open/close presentation connection
- Send/receive presentation connection messages

# Presentation URL Availability

```
presentation-url-availability-request = {  
  request  
  1: int ; watch-id  
  2: microseconds ; watch-duration  
  3: [* url] ; urls  
}
```

```
url = text
```

```
; idea: use HTTP response codes?
```

```
url-availability = &(  
  not-compatible: 0  
  compatible: 1  
  not-valid: 10  
)
```

```
presentation-url-availability-response = {  
  response  
  1: [* url-availability] ; url-availabilities  
}
```

```
; Send when availability changes as  
; long as requester is interested  
; Not broadcast!
```

```
presentation-url-availability-event = {  
  1: int ; watch-id  
  2: [* url] ; urls  
  3: [* url-availability] ; url-availabilities  
}
```

# Presentation Initiation

```
presentation-initiation-request = {  
  request  
  1: presentation-id ; presentation-id  
  2: url ; url  
  3: text ; headers  
  ; opens a connection at the same time  
  4: connection-id ; connection-id  
}
```

; text because of <https://w3c.github.io/presentation-api/#dfn-presentation-identifier>.

```
presentation-id = text
```

```
; Don't send response until URL is loaded  
presentation-initiation-response = {  
  response  
  1: &result ; result  
  ; TODO: Add optional HTTP response code?  
}
```



# Presentation Termination

```
presentation-termination-request = {  
  request  
  1: presentation-id ; presentation-id  
  ; idea: split up into  
  ; 1 enum for (user, not)  
  ; 1 enum for event type  
  2: &(  
    controller: 10  
    user-via-controller: 11  
  ) ; reason  
}
```

```
presentation-termination-response = {  
  response  
  1: result ; result  
}  
  
presentation-termination-event = {  
  1: presentation-id ; presentation-id  
  2: &(  
    receiver: 1  
    user-via-receiver: 2  
    controller: 10  
    user-via-controller: 11  
    new-replacing-current: 20  
    idle-too-long: 30  
  ) ; reason  
}
```

# Presentation Connection open/close

```
presentation-connection-open-request = {  
  request  
  1: presentation-id ; presentation-id  
  2: connection-id ; connection-id  
}
```

```
presentation-connection-close-request = {  
  request  
  1: connection-id ; connection-id  
}
```

connection-id = uint

; connection-id scoped across presentations but

; not across device

; Receiver needs to keep map of

; (device, connection-id) => presentation-id

; to demux

```
presentation-connection-open-response = {  
  response  
  1: &result ; result  
}
```

```
presentation-connection-close-response = {  
  response  
  1: &result ; result  
}
```

```
presentation-connection-close-event = {  
  1: connection-id ; connection-id  
  2: &(close-method, navigation, page-gone, weird-error) ;  
  ? 3: text ; error-message  
}
```

# Presentation Connection messages

```
presentation-connection-message = {  
  1: connection-id ; connection-id  
  2: bytes / text ; message  
}
```

# Remote Playback API messages

- Remote playback availability
- Start/stop remote playback
- Remote playback controls (pause, volume, etc.)
- Remote playback status

# Remote Playback Availability

```
remote-playback-availability-request = {  
  request  
  1: int ; watch-id  
  ; To renew interest, resend request  
  2: microseconds ; watch-duration  
  3: [* url] ; urls  
}
```

```
url = text
```

```
; Idea: use HTTP response codes?
```

```
url-availability = &  
  not-compatible: 0  
  compatible: 1  
  not-valid: 10  
)
```

```
remote-playback-availability-response = {  
  response  
  url-availabilities: [* url-availability]  
}
```

```
; Send when availability changes as  
; long as requester is interested  
; Not broadcast!
```

```
remote-playback-availability-event = {  
  1: int ; watch-id  
  2: [* url] ; urls  
  3: [* url-availability] ; url-availabilities  
}
```

Needs impl feedback about whether URL availability is sufficient for the use case.

Question: should we unify with presentation URL availability?

# Start/stop remote playback

```
remote-playback-start-request = {  
  request  
  1: remote-playback-id ; remote-playback-id  
  2: [* url] ; urls  
  ? 3: [* url] ; text-track-urls  
  4: remote-playback-controls ; controls  
}
```

```
remote-playback-stop-request = {  
  request  
  1: remote-playback-id ; remote-playback-id  
}
```

```
remote-playback-id = uint
```

```
remote-playback-start-response = {  
  response  
  1: &result ; result  
  ? 2: remote-playback-state ; state  
}
```

```
remote-playback-stop-response = {  
  response  
  1: &result ; result  
}
```

```
remote-playback-stop-event = {  
  1: remote-playback-id ; remote-playback-id  
  ; TODO: reasons  
}
```

MISSING: <source> extended mime type, media query (replace url here)

# Remote playback controls

Things we need to support changing:

- set src
- set preload (none, metadata, auto): optional to implement
- play
- pause
- set currentTime (seek)
- set playbackRate (slow-mo, fast-forward, rewind): optional to implement
- set loop (play at start when end is reached)
- set volume (0.0-1.0)
- set muted
- set poster image (when no data available): optional to implement
- enable/disable audio tracks, video tracks: optional to implement
- enable/disable text tracks
- add/remove text track cues: optional to implement

# Remote playback controls

```
remote-playback-modify-request = {
  request
  1: remote-playback-id ; remote-playback-id
  2: remote-playback-controls ; controls
)
remote-playback-controls = {
  ? 1: url ; source
  ? 2: bool ; preload
  ? 3: bool ; loop
  ? 4: bool ; paused
  ? 5: bool ; muted
  ? 6: float ; volume
  ? 7: media-time ; seek
  ? 8: media-time ; fast-seek
  ? 9: float ; playback-rate
  ? 10: url ; poster
  ? 11: [* bool] ; audio-tracks-enabled
  ? 12: uint ; enabled-video-track-index
  ? 13: [* text-track-mode] ; text-tracks-mode
}
text-track-mode = &(disabled / showing / hidden)
```

```
remote-playback-modify-response = {
  response
  1: &result ; result
  ? 2: remote-playback-state ; state
}
remote-playback-text-track-add-cue = {
  1: remote-playback-id ; remote-playback-id
  2: text ; track-id
  3: cue ; text-track-queue
}
remote-playback-text-track-remove-cue = {
  1: remote-playback-id ; remote-playback-id
  2: text ; track-id
  3: cue ; text-track-queue
}
text-track-queue = {
  1: text ; id
  2: media-time-range ; range
  3: any ; payload
}
```



# Remote playback state updates

```
; TODO: frequency to update current-time
; HTMLMediaElement does every 250ms?
remote-playback-state-event = {
  1: remote-playback-id ; remote-playback-id
  2: remote-playback-state ; state
}

media-error = [
  code: &(...)
  message: text
]

media-track-state = {
  1: text ; label
  2: text ; language
  ; MISSING: for text tracks: cues and activeCues
}
```

```
remote-playback-state = {
  ? 1: url ; current-source
  ? 2: &(...) ; network-state
  ? 3: &(...) ; ready-state
  ? 4: (rate: bool, preload: bool, poster-image: bool) ; supports
  ? 5: bool ; paused
  ? 6: bool ; ended
  ? 7: microseconds ; timeline-offset
  ? 8: media-time ; current-time
  ? 9: media-duration ; duration
  ? 10: [* media-time-range] ; buffered-time-ranges
  ? 11: [* media-time-range] ; played-time-ranges
  ? 12: [* media-time-range] ; seekable-time-ranges
  ? 13: video-resolution ; resolution
  ? 14: float ; volume
  ? 15: bool ; muted
  ? 16: float : playbackRate
  ? 17: media-error : error
  ? 18: bool ; seeking
  ? 19: bool ; stalled
  ? 20: [* media-track-state] audio-tracks
  ? 21: [* media-track-state] video-tracks
  ? 22: [* media-track-state] data-tracks
}
```

# Remote playback types

```
media-time = [  
  value: uint  
  scale: uint  
]
```

```
media-duration = media-time / unknown / forever  
unknown = 0;  
forever = 1
```

```
media-time-range = [  
  start: media-time,  
  end: media-time  
]
```

# Message ordering

Generally:

1. Requests that go together have their own group  
(presentation API, remote playback API)
2. Responses can be out of order, except responses that have update events  
(then put them all in one group)
3. Some things are special (audio/video frames, presentation connection messages)

# Questions to answer

- ~~Go with DNS-SD services `_openscreen._udp.local`?~~
- ~~Which TXT metadata? version, device ID, friendly name, model name, type, capabilities~~
- ~~When to keep QUIC connections alive? To receive?~~
- ~~How to keep alive? With OSP-level status every 25s?~~
- ~~How to map to QUIC streams? CBOR back-to-back when ordered messages needed?~~
- ~~Use CDDL?~~
- ~~Use CBOR tags or out-of-band tags?~~
- ~~Use mostly string keys in CBOR (others when bits are precious)?~~
- ~~Use JPAKE after TLS, in TLS, or not at all?~~
- ~~Use the Presentation API message as proposed?~~
- ~~Use the Remote Playback API messages as proposed?~~

# Open Screen Protocol

## Day 2

---

Peter Thatcher ([pthatcher@google.com](mailto:pthatcher@google.com))

Mark Foltz ([mfoltz@google.com](mailto:mfoltz@google.com))

TPAC 2018

# Day 2 - Outline

Open Screen Library

Security and Privacy

Off-LAN support with ICE

Support for media streaming

2019 goals and planning

# Open Screen Library

- Objectives
- Architecture
- Roadmap
- Status and next steps
- Contributing



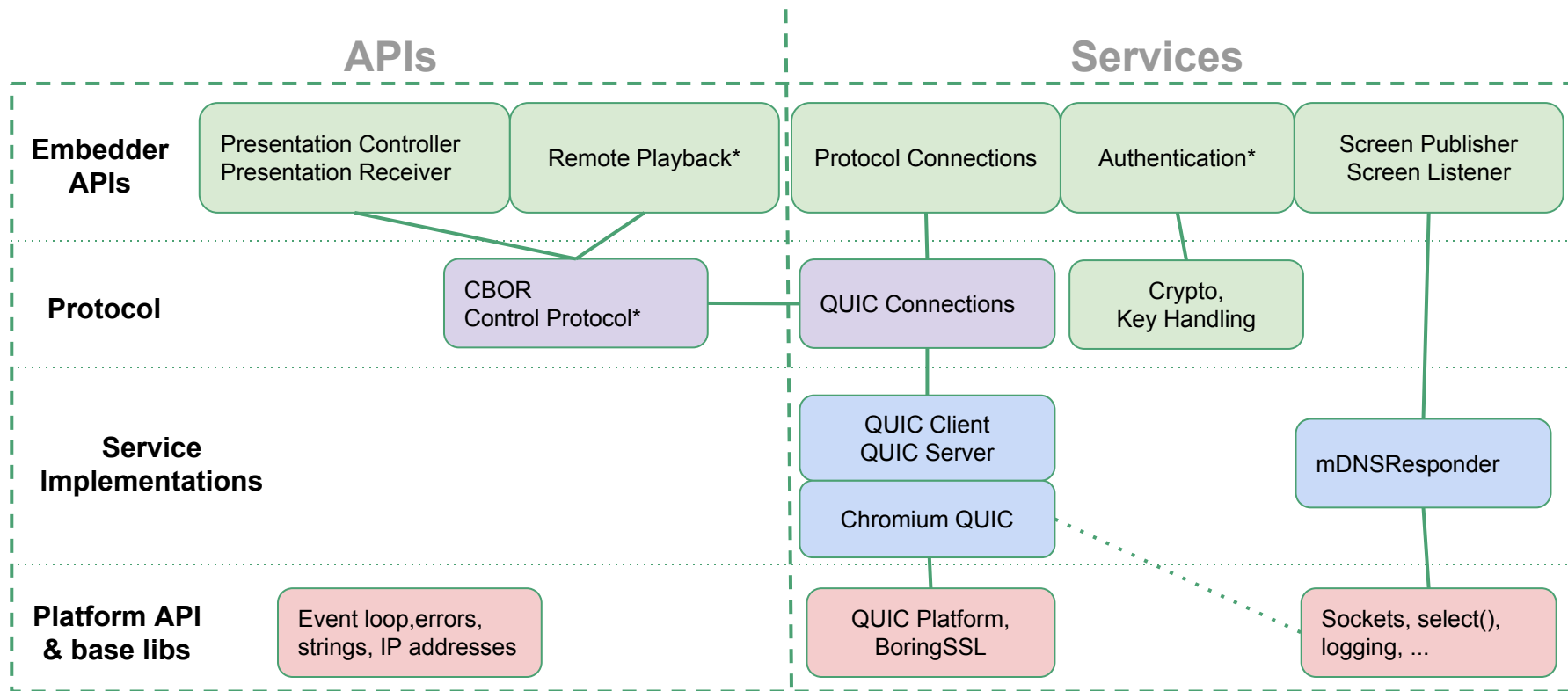
<https://chromium.googlesource.com/openscreen/>

# Open Screen Library: Objectives

1. Free and open source (Chromium license)
2. Self contained; embeddable; OS abstraction layer
3. Small footprint (binary and memory size); YAGNI principle
4. Presentation: controller/receiver, Remote Playback: local/remote
5. Embedder is responsible for rendering HTML5 and media
6. Extensible with new features / protocols



# Open Screen Library: Architecture



\* To be implemented

# Demo!

[https://drive.google.com/open?id=17rdnVmt6thobWtmVZMG37PiKOp\\_kWGat](https://drive.google.com/open?id=17rdnVmt6thobWtmVZMG37PiKOp_kWGat)

# Open Screen Library: Presentation Controller

```
class openscreen::presentation::Controller {  
    ScreenWatch RegisterScreenWatch(const std::string& url, ScreenObserver* observer);  
  
    ConnectRequest StartPresentation(const std::string& url, const std::string& screen_id,  
                                     RequestDelegate* delegate, Connection::Delegate* conn_delegate);  
  
    ConnectRequest ReconnectPresentation(const std::string& presentation_id, const std::string& screen_id,  
                                           RequestDelegate* delegate, Connection::Delegate* conn_delegate);  
  
    void OnPresentationTerminated(const std::string& presentation_id, TerminationReason reason);  
};
```

RequestDelegate receives the result of the request (including a new Connection).

Connection::Delegate receives callbacks from the new Connection.

# Open Screen Library: Screen Publisher

```
class openscreen::ScreenPublisher {  
    virtual bool Start() = 0;  
    virtual bool StartAndSuspend() = 0;  
    virtual bool Stop() = 0;  
    virtual bool Suspend() = 0;  
    virtual bool Resume() = 0;  
  
    virtual void UpdateFriendlyName(const std::string& friendly_name) = 0;  
    State state();  
    ScreenPublisherError last_error() const { return last_error_; }  
};
```

Services can be controlled by the embedder (for efficiency/power).

Separate Observer object gets callbacks with metrics, errors, and state changes.

# Open Screen Library: CDDL code generation

```
request = (  
  request-id: request-id  
)
```

```
presentation-url-availability-request = {  
  request  
  urls: [* url]  
}
```

# Open Screen Library: Platform API

```
UdpSocketPtr CreateUdpSocketIPv4();
```

```
UdpSocketPtr CreateUdpSocketIPv6();
```

```
void DestroyUdpSocket(UdpSocketPtr socket);
```

```
bool BindUdpSocket(UdpSocketPtr socket,  
                  const IPEndpoint& endpoint,  
                  int32_t ifindex);
```

```
bool JoinUdpMulticastGroup(UdpSocketPtr socket,  
                            const IPAddress& address,  
                            int32_t ifindex);
```

# Open Screen Library: Platform API

```
UdpSocketPtr CreateUdpSocketIPv4();
```

```
UdpSocketPtr CreateUdpSocketIPv6();
```

```
void DestroyUdpSocket(UdpSocketPtr socket);
```

```
bool BindUdpSocket(UdpSocketPtr socket,  
                  const IPEndpoint& endpoint,  
                  int32_t ifindex);
```

```
bool JoinUdpMulticastGroup(UdpSocketPtr socket,  
                           const IPAddress& address,  
                           int32_t ifindex);
```

```
int64_t ReceiveUdp(UdpSocketPtr socket,  
                   void* data,  
                   int64_t length,  
                   IPEndpoint* src,  
                   IPEndpoint* original_destination);
```

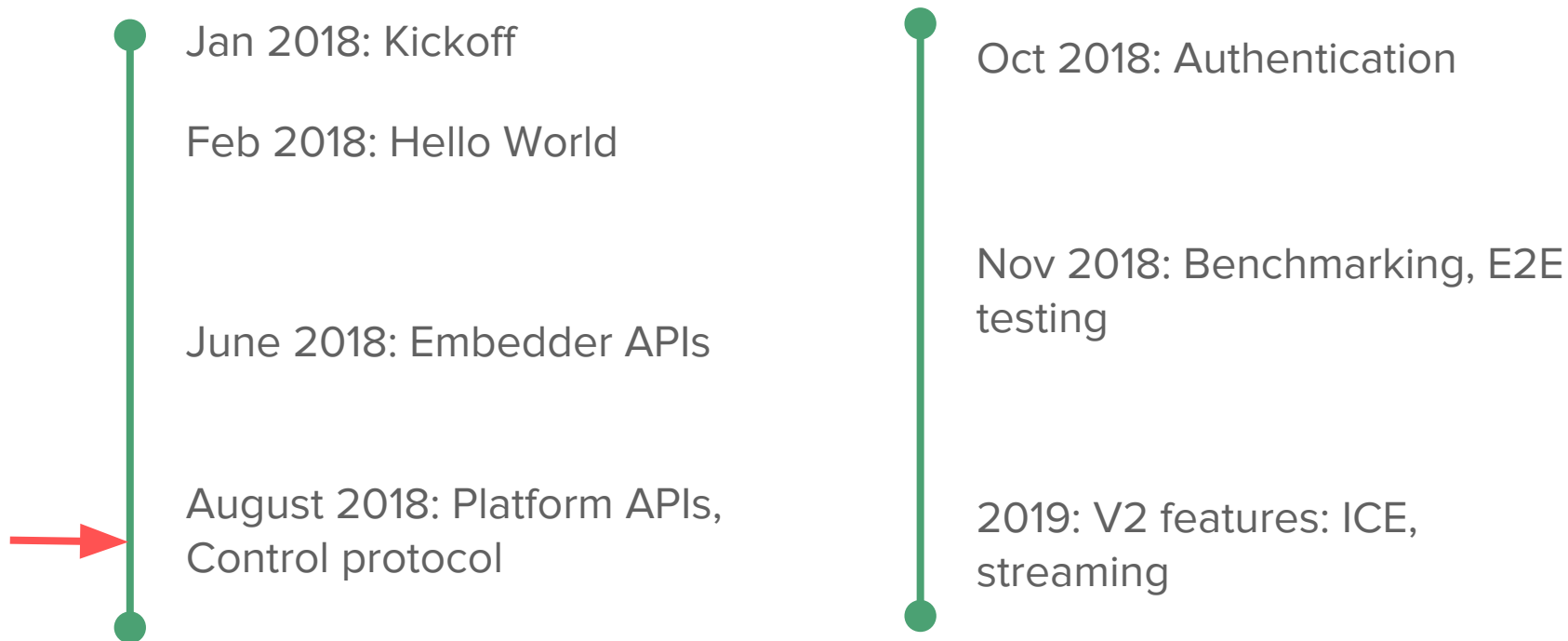
```
int64_t SendUdp(UdpSocketPtr socket,  
                const void* data,  
                int64_t length,  
                const IPEndpoint& dest);
```

# Open Screen Library: CDDL code generation

```
ssize_t EncodePresentationUrlAvailabilityRequest(const PresentationUrlAvailabilityRequest& data, uint8_t* buffer, size_t length) {
    CBOR_RETURN_ON_ERROR(cbor_encoder_create_map(&encoder0, &encoder1, 2));
    CBOR_RETURN_ON_ERROR(cbor_encode_text_string(&encoder1, "request-id", sizeof("request-id") - 1));
    CBOR_RETURN_ON_ERROR(cbor_encode_uint(&encoder1, data.request_id));
    CBOR_RETURN_ON_ERROR(cbor_encode_text_string(&encoder1, "urls", sizeof("urls") - 1));
    CBOR_RETURN_ON_ERROR(cbor_encoder_create_array(&encoder1, &encoder2, data.urls.size()));
    for (const auto& x : data.urls) {
        if (!IsValidUtf8(x)) {
            return -CborErrorInvalidUtf8TextString;
        }
        CBOR_RETURN_ON_ERROR(cbor_encode_text_string(&encoder2, x.c_str(), x.size()));
    }
    CBOR_RETURN_ON_ERROR(cbor_encoder_close_container(&encoder1, &encoder2));
    CBOR_RETURN_ON_ERROR(cbor_encoder_close_container(&encoder0, &encoder1));
    ...
}
```



# Open Screen Library: Roadmap



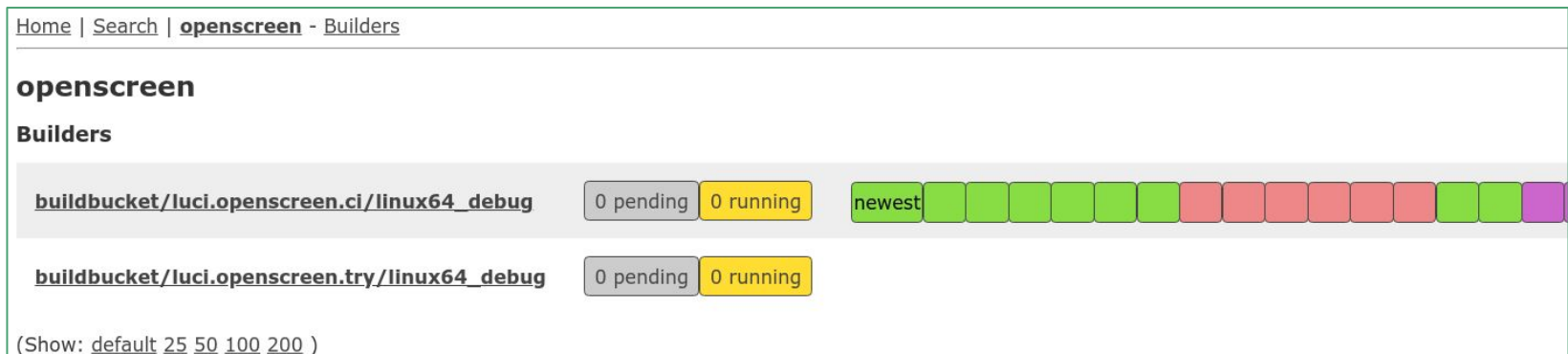
# Open Screen Library: Status and Next Steps

- CDDL-based code generation for parse/serialize
  - Will handle all messages proposed here
- Support for Mac OS X development (currently Linux-only)
- Bring up end to end on Raspberry Pi
- Close to end to end demo .... design choices will determine:
  - CBOR message framing
  - QUIC connection handling
  - Authentication handshake

# Open Screen Library: Contributing

Closer to enabling external contributions.

Finalizing infrastructure (continuous build, try jobs, submit queue).




Home | Search | **openscreen** - Builders

---

**openscreen**

**Builders**

<a href="#">buildbucket/luci.openscreen.ci/linux64_debug</a>	0 pending 0 running	newest	
<a href="#">buildbucket/luci.openscreen.try/linux64_debug</a>	0 pending 0 running		

(Show: [default](#) [25](#) [50](#) [100](#) [200](#) )

Get the code: <https://chromium.googlesource.com/openscreen/>

# Open Screen Protocol: Privacy & Security

Questions to consider:

- What information handled by Open Screen Protocol should be protected?
- What are threat models to consider?
- Which threats can be addressed by protocol design?
- What are additional kinds of mitigations?

Considerations: <https://www.w3.org/TR/security-privacy-questionnaire/>

# Personally Identifiable Information & High Value Data

- Presentation URLs and IDs
- Presentation messages
- Remote playback URLs
- Streaming media content
- Private keys and J-PAKE passwords

**This information is a priority to protect.**

However, presentation connection data may be advertised (kiosk/sign)

# Device specific data

- Device GUIDs, model names, friendly names
- Device capabilities
- Compatibility with presentation URLs/remote playback URLs
- Public keys

Network exposes device identifiers on the LAN (via MAC addresses).

Protocol can be used to query for device capabilities.

**Device data reveals identity, capabilities of device.**

**Friendly name may be PII.**

# Threat model: Passive attacker

- Gains access to LAN (compromised WiFi security, compromised client or router)
- Able to observe but not modify network traffic.
- Learns device data advertised via mDNS
- Observes connections and can distinguish parties involved

**Mitigation: Minimize data exposed prior to establishment of an encrypted transport.**

Note: Passive attacks are possible on the handshake and encrypted stream.

In addition to proper crypto implementation, implications for key rotation.



# Threat model: Active attacker

- Gains access to LAN (compromised WiFi security, client, or router)
- Able to modify/inject network traffic

**Attack #1:** Impersonation ("MITM") of receiver

**Attack #2:** Impersonation of controller

**Attack #3:** Denial of service





# Attack #1: Impersonation of receiver

## Mitigations:

- Out of band verification of receiver identity by human (password, QR code)
- Third party assertion of identity (e.g., key signing)
- Flag less trusted metadata in the controller UI
- Detect impersonation attempts: (MAC, key FP, IP:port, name) collisions
- Lose trust on change of identity



Homer? Who is Homer? My name is Guy Incognito.

## Attack #2: Impersonation of controller

Protocol allows controllers to resume connections to existing presentations/remote playbacks.

Mitigations:

- Require a client certificate for controllers in TLS handshake
- Harden or extend IDs used to reconnect to presentations/remote playbacks.
- Don't expose user data without a certificate and a valid ID



# Attack #3: Denial of Service

- Attempts to block mDNS by publishing fake records.
- Attempts to deny connections to a receiver by repeatedly attempting new connections/handshakes.

## Mitigations:

- mDNS suggests DNSSEC to allow filtering of mDNS responses.
- Hard limits on number of connection attempts per 5-tuple.



**Needs more investigation**, but may be infeasible to prevent fully on a LAN.

# Other mitigations (beyond protocol design)

- Platform/OS security
  - Hardware backed crypto
  - Verified boot
- Rendering engine security
  - Sandboxing untrusted code/content
  - Origin isolation
- Software updates
  - Keeping security relevant libraries patched
- Developer education, user interface guidelines
- Key management and control

# Next steps

- Complete [Security & Privacy questionnaire](#) and include in [V1 spec](#).
- Continue to research TLS and J-PAKE best practices.
- Consider how we might evolve security architecture.
- Additional reviews (internal & W3C).
- Feedback from application developers.

# Things left to discuss for OSP

- Review of where we're at from yesterday
- JPAKE
- ICE
- Streaming
- WebRTC polyfill

# Where we're at from yesterday

- Got most of the basics of the protocol figured out (with resolutions)
  - Unresolved: length-prefixing vs. "back to back"
  - Unresolved: type-prefixing vs. CBOR tagging
  - Still need to talk about JPAKE
- Got most of Presentation API figured out (w/o specific resolutions)
  - TODO: the various reason enums
- Got most (?) or Remote Playback API figured out (w/o specific resolutions)
  - Unresolved: <source> extended mime types, media queries
  - Unresolved: receiver sending text track cues back to controller

# JPAKE

Discussion pushed to today from yesterday

3 options:

A: JPAKE after TLS

B: JPAKE replaces TLS

C: non-JPAKE after TLS



# JPAKE after TLS, 2-RTT variant, first RTT

```
jpake-handshake-start-request = {  
  request  
  1: bytes ; signer-id (fingerprint)  
  
  ; Start round 1; maybe include g, p, q  
  ; x1 is random in (0, q-1)  
  ; gx1 = g^x1 mod p  
  4: bignum ; gx1  
  5: zero-knowledge-proof ; proof-of-x1  
  ; x2 is random in (1, q-1)  
  ; gx2 = g^x2 mod p  
  6: bignum ; gx2  
  7: zero-knowledge-proof ; proof-of-x2  
}
```

```
jpake-handshake-start-response = {  
  request  
  1: bytes ; signer-id (fingerprint)  
  
  ; finish round 1  
  ; x3 is random in (0, q-1)  
  ; gx3 = g^x3 mod p  
  4: bignum ; gx3  
  5: zero-knowledge-proof ; proof-of-x3  
  ; x4 is random in (1, q-1)  
  ; gx4 = g^x4 mod p  
  6: bignum ; gx4  
  7: zero-knowledge-proof ; proof-of-x4  
  
  ; start round2  
  ; b = g^((x1+x2+x3)*x4*shared_secret)  
  8: bignum ; b  
  ; x4s = x4 * shared_secret  
  9: zero-knowledge-proof ; proof-of-x4s  
}
```

# JPAKE after TLS, 2-RTT variant, second RTT

```
jpake-handshake-finish-request = {  
  request  
  
  ; finish round 2  
  ;  $a = g^{((x1+x3+x4)*x2*shared\_secret)}$   
  1: bignum ; a  
  ;  $x2s = x2 * shared\_secret$   
  2: zero-knowledge-proof ; proof-of-x2s  
  
  ; start round 3  
  ; sha256(sha256(key))  
  3: bytes ; double-hashed-key  
}
```

```
jpake-handshake-finish-response = {  
  response  
  
  ; finish round 3  
  ; sha256(key)  
  3: bytes ; hashed-key  
}
```

# JPAKE zero knowledge proof

```
; See RFC 8235 section 2.2
zero-knowledge-proof = {
  ; v is random in (0, q-1)
  ;  $gv = g^v \pmod p$ 
  0: bignum ; gv

  ;  $r = v - \text{proven\_value} * \text{challenge} \pmod q$ 
  ; challenge is sha1(g ||  $g^v$  ||  $g^{\text{proven\_value}}$  || signer ID)
  ; (where || is string concatenation)
  ; (see crypto/jpake/jpake.c)
  1: bignum ; r
}
```

# Option B: JPAKE replaces TLS handshake

Something like IETF [draft-cragie-tls-ecjpake](#).

It's not clear whether it can happen or not, though. There are issues.

If it does happen, we can do this:

1. First connection uses JPAKE has handshake
2. Using first connection, swap TLS certificates and cache them
3. Subsequent connections use TLS with cached certificates

# Option C: challenge/response after TLS

JPAKE does shared secret → keys + mutual authentication

TLS produces keys

Challenge/response after TLS does shared secret → mutual authentication

So, TLS + challenge/response does shared secret → keys + mutual authentication

## Option C: "challenge/response" after TLS

```
authentication-request = {  
  request  
  
  1: bytes ; challenge  
}  
  
authentication-response = {  
  response  
  
  ; fingerprints to avoid reflection/MITM attacks  
  ; sha256(challenge || password  
  ;           || challenger-fp || responder-fp)  
  ; where || is string concatenation  
  1: bytes ; authentication  
}
```



# JPAKE options comparison

## JPAKE replaces TLS

- hardest to spec and implement
- easiest to get wrong
- 2-RTT first connection

## JPAKE after TLS

- easier to spec and implement
- harder to get wrong
- 3-RTT first connection

## "challenge/response" after TLS

- very easy to implement
- hard to get wrong (assuming we get security review that the scheme works)
- 2-RTT first connection

# JPAKE recommendations/resolution

## Recommendations:

- Don't do "JPAKE replace TLS"
- Consider dropping JPAKE for challenge/response
- If we don't want to drop JPAKE, do "JPAKE after TLS"



# Quick intro ICE :-)

- Converts a messaging/signaling channel into a p2p data channel by trying lots of network paths called “ICE candidate pairs”
- To do its work, ICE must have a way to exchange two kinds of information:
  - ICE session parameters (basically a session ID and password)
  - ICE candidates (basically IP+port combos)
- Some things to know:
  - STUN is a way to get your public IP+port on the other side of a NAT
  - TURN is a way to relay data through a server
  - STUN and TURN servers don't need to be shared by endpoints. They can be different servers that don't know about each other.
  - Either side can initiate ICE, but if both initiate at the same time, you can actually just respond with the same session parameters as your request and it will keep working (because role conflicts can be resolved within ICE itself).



# ICE messages <https://github.com/webscreens/openscreenprotocol/issues/73>

```
ice-session-start-request = {  
  request  
  
  1: uint ; generation  
  2: string ; username-fragment  
  3: string ; password  
  4: &(amp;controlling=1, controlled=2) ; role  
  5: [* string] ; options  
}
```

```
ice-session-start-response = {  
  response  
  
  1: string ; username-fragment  
  2: string ; password  
  3: [* string] ; options  
}
```

```
ice-candidate-added = {  
  0: uint ; id  
  1: (bytes .size 4) / (bytes .size 16) ; ip  
  2: uint ; port  
  3: &(amp;host, server-reflexive, turn, tcp, tls) ; type  
  4: uint ; priority  
}
```

```
ice-candidates-removed = {  
  0: uint ; generation  
  1: [* uint] ; candidate-ids  
}
```

# ICE in Web API

## Option A: RTCIceTransport

1. App gets RTCIceTransport via its own signaling/discovery
2. App passes it to new Remote Playback or Presentation API
3. Browser creates QUIC connection over IceTransport
4. Browser takes it from there

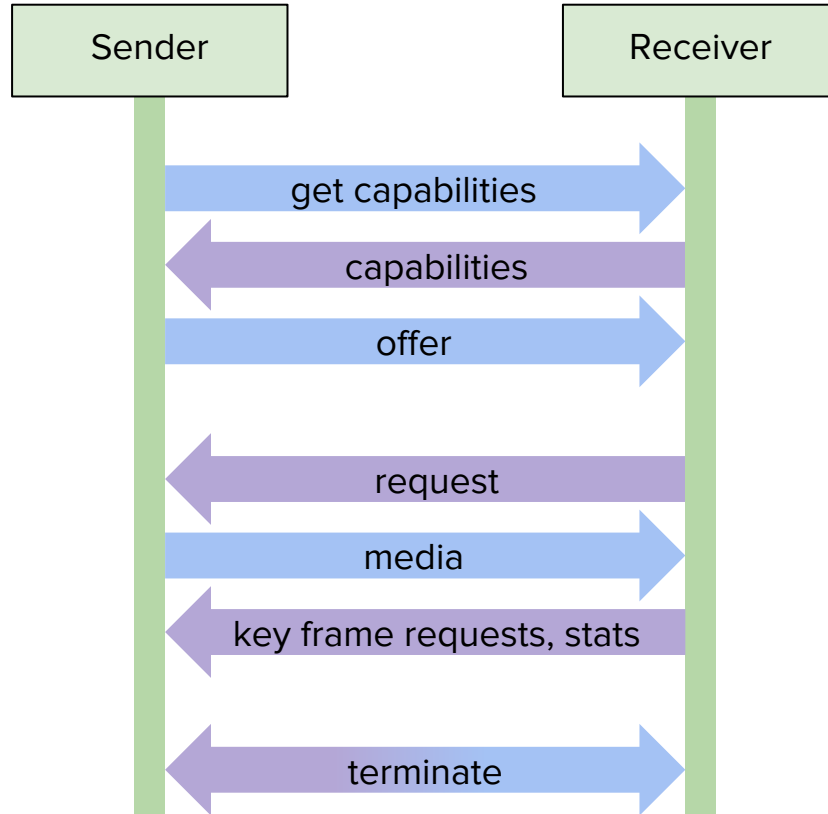
## Option B: RTCQuicTransport

1. App gets RTCQuicTransport via its own signaling/discovery (maybe c2s)
2. App passes it to new Remote Playback or Presentation API
3. Browser takes it from there

# Streaming messages

- Get receiver capabilities
- Offer/request streams
- **Send audio/video/data**
- Request video key frame
- Send stats

# Media Streaming Flow



# Audio Frames

```
; Using array structure and integer keys to save bytes
audio-frame = [
  encoding-id: uint    ; gives implicit clockrate/timescale, duration, and codec
  start-time: uint     ; in terms of clockrate/timescale
                      ; end-time = start-time + duration
  payload: bytes       ; interpreted according to codec type
  ? optional: {
    ; Maybe add this in the future? ? 0: uint           ; frame-id or sequence-number
    ? 1: uint        ; duration defaults to encoding's value
    ? 2: media-time  ; system-time for synchronization
  }
]
```

# Video Frames

```
video-frame = {  
  0: uint ; encoding-id: gives implicit clockrate/timescale, duration, and codec  
  1: uint ; frame-id ; because durations are often unknown; decode order, not presentation order  
  2: uint ; start-time in terms of encoding clockrate/timescale  
  3: any ; payload codec-specific, typically bytes  
    ; Missing means depends on (frame-id - 1)  
    ; Is present and empty ([]), then it's a key frame  
  ? 4: [* uint] ; depends-on frame IDs ; needs encoding id for SVC  
  ? 5: uint ; duration  
        ; end-time = start-time + duration  
        ; defaults to encoding's value  
  ? 6: video-rotation ; rotation; default to encodings's value  
  ? 7: media-time ; system-time for synchronization  
}
```

TODO: can we have the rule for missing be "some frame" that the codec can handle, not just "-1"?

TODO: do we need a field for "a future frame will depend on this one"?

TODO: examples of codec-specific payloads; change to bytes if we can't think of one

# Data Frames

```
data-frame = [  
  encoding-id: uint  
  payload: any      ; format-specific, typically bytes  
  ? metadata : {  
    ? 0: uint      ; frame-id or sequence-number  
    ? 1: uint      ; timestamp in terms of encoding's clockrate/timescale  
    ? 2: media-time ; system-time for synchronization  
  }  
]
```

TODO: Come up with examples of typical data that would go along with audio/video (text tracks, mouse pointer, AR/VR)



# Media Streaming Capabilities

(Learn what the other side can do)

```
streaming-capabilities-request {  
  request  
}
```

```
streaming-capabilities-response {  
  response  
  capabilities: streaming-capabilities  
}
```

```
streaming-capabilities = {  
  receive-audio: [* receive-audio-capability]  
  receive-video: [* receive-video-capability]  
  receive-data: [* receive-data-capability]  
}
```

```
receive-audio-capability = {  
  format: text ; codec/mimetype  
  format-parameters: any ; codec-specific  
}
```

```
receive-video-capability = {  
  format: text ; codec/mimetype  
  format-parameters: any ; codec-specific  
  ? max-resolution: video-resolution  
  ? max-frames-per-second: float  
  ? max-pixels-per-second: float  
  ; ... plenty of room for expansion  
}
```

```
receive-data-capability = {  
  format: text  
  format-parameters: any ; format-specific  
}
```

# Streaming Offers

```
streaming-offer = {  
  streams: [* media-stream]  
  ? audio: [* audio-encoding-offer]  
  ? video: [* video-encoding-offer]  
  ? data: [* data-encoding-offer]  
}
```

```
media-stream = {  
  id: uint  
  ? friendly-name: text  
}
```

```
encoding-offer = {  
  stream-id: uint  
  encoding-id: uint  
  format: text ; codec/mimetype  
  format-parameters: any ; codec-specific  
}
```

```
audio-encoding-offer = {  
  encoding-offer  
  channel-count: uint ; may vary by frame;  
  sample-rate-hz: uint  
  sample-count: uint ; may vary by frame  
}
```

```
video-encoding-offer = {  
  encoding-offer  
  duration: uint ; may vary by frame  
  clock-rate: uint ; probably want 90khz  
  ; ... rotation, dependencies for SVC  
}
```

```
data-encoding-offer = {  
  encoding-offer  
  clock-rate-hz: uint ; default is 1Mhz  
}
```

# Streaming Requests

```
streaming-request = {  
  request  
  audio: [* audio-encoding-request]  
  video: [* video-encoding-request]  
  data: [* data-encoding-request]  
}
```

```
streaming-response = {  
  response  
}
```

```
audio-stream-request = {  
  encoding-id: uint  
  ? max-bits-per-second: uint ; default is unlimited  
}
```

```
video-stream-request = {  
  encoding-id: uint  
  ? max-resolution: video-resolution ; default is capability  
  ? max-frames-per-second: float ; default is capability  
  ? max-bits-per-second: uint ; default is capability  
}
```

```
data-stream-request = {  
  encoding-id: uint  
}
```

# Key frame requests

A higher request ID invalidates all previous ones. If the last decode frame ID is set, the sender knows to provide a key frame to get past that point (or make a delta frame from that point). If not set, send anything.

```
video-key-frame-request = {  
  encoding-id: uint  
  request-id: uint  
  ? last-decoded-frame-id: uint  
}
```

# Streaming Stats

```
streaming-stats-request = {  
  request  
  interval: microseconds  
}
```

```
streaming-stats-response = {  
  response  
  result: &result  
}
```

```
streaming-receiver-stats-event = {  
  timestamp: microseconds  
  audio: [* {  
    encoding-id: uint  
    ? received: microseconds  
    ? lost: microseconds  
    ? buffer-delay: microseconds  
    ? decode-delay: microseconds  
    ; ...  
  }]  
  video: [* {  
    encoding-id: uint  
    ? received: microseconds  
    ? lost: microseconds  
    ? buffer-delay: microseconds  
    ? decode-delay: microseconds  
  }]  
}
```

```
streaming-sender-stats-event = {  
  timestamp: microseconds  
  audio: [* {  
    encoding-id: uint  
    ? sent: microseconds  
    ? encode-delay: microseconds  
    ; ...  
  }]  
  video: [* {  
    encoding-id: uint  
    ? encode-delay: microseconds  
    ? sent-frames: uint  
    ? dropped-frames: uint  
  }]  
}
```

# RTC Polyfill: APIs we could use

What we could build on (assuming they get implemented):

- RTCIceTransport
- RTCQuicTransport

What we cannot build on (there are no web APIs):

- mDNS
- User dialog for receiver selection

# RTC Polyfill: things we can do

What we could do:

- Use something for discovery instead of mDNS (and ICE for connectivity)
- Implement OSP in JS (CBOR, JPAKE, Presentation API messages, etc)

What we could not do:

- use mDNS for discovery
- Cause the browser's receiver selection dialog to pop up

# WebRTC Polyfill: User experience

The user experience would be like this:

1. UX for selecting a receiver would **show up in the page**, not in the browser
2. Endpoints **discoverable through** (not mDNS) would show up in the UX

The big limitation is: how are devices on the local network discovered?

It would require the devices to register with **some server-based discovery system**



# WebRTC Polyfill: Streaming

To do streaming, we would also need:

- audio/video capture
  - already there? or coming soon?
- encoders
  - not strictly needed, esp for audio (opus or vp8 in wasm/js!)
  - buried in RTCRtpSender/RTCRtpReceiver; WebRTC WG looking at ways to make it accessible

End of prepared slides

---

# Data exposed cross origin

- Presentation URL and remote playback availability is exposed cross origin.
- Presentation messages are cross origin.

**Open Screen Protocol does not convey proof of origin or restrict cross-origin.**

**Note these capabilities are inherent in the design of the relevant APIs.**