Open Screen Protocol
Day 1

Peter Thatcher (pthatcher@google.com)
Mark Foltz (mfoltz@google.com)
Berlin May 2019

Day 1 - Outline

Introductions, Agenda Bashing

Overview of Group Work

Review of Open Screen Protocol 1.0

OSP Authentication & Security

OSP Protocol Design Issues

Presentation API Protocol Items

Remote Playback API Protocol Items

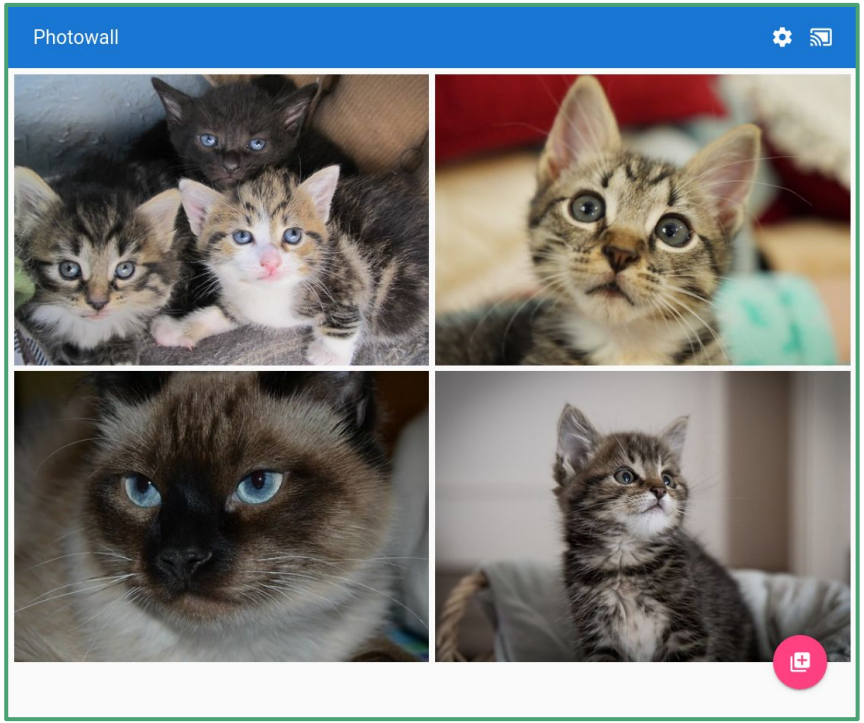
Second Screen CG History

- 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
- Jan-Feb 2018: SSWG rechartered. Phone conference, work plan
- May 2018: Berlin F2F
- October 2018: TPAC 2018
- April 2019: 1.0 draft spec released
- May 2019: Here we are :-)

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.

Presentation API - In practice

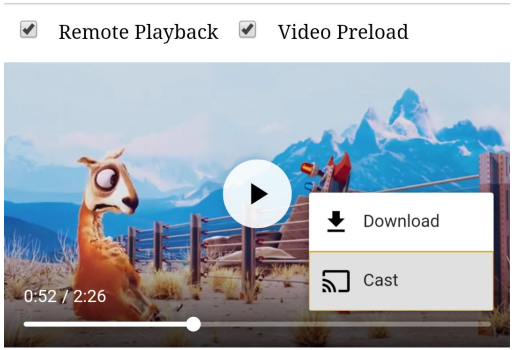


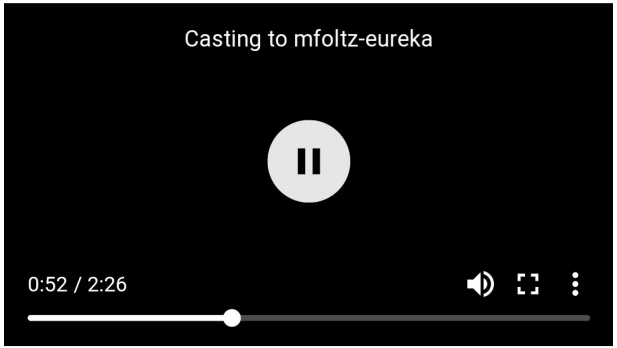
https://googlechromelabs.github.io/presentation-api-samples/photowall/

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 state is synchronized with the remote playback device





Second Screen Community Group Scope

Address **interoperability** through **protocol incubation**

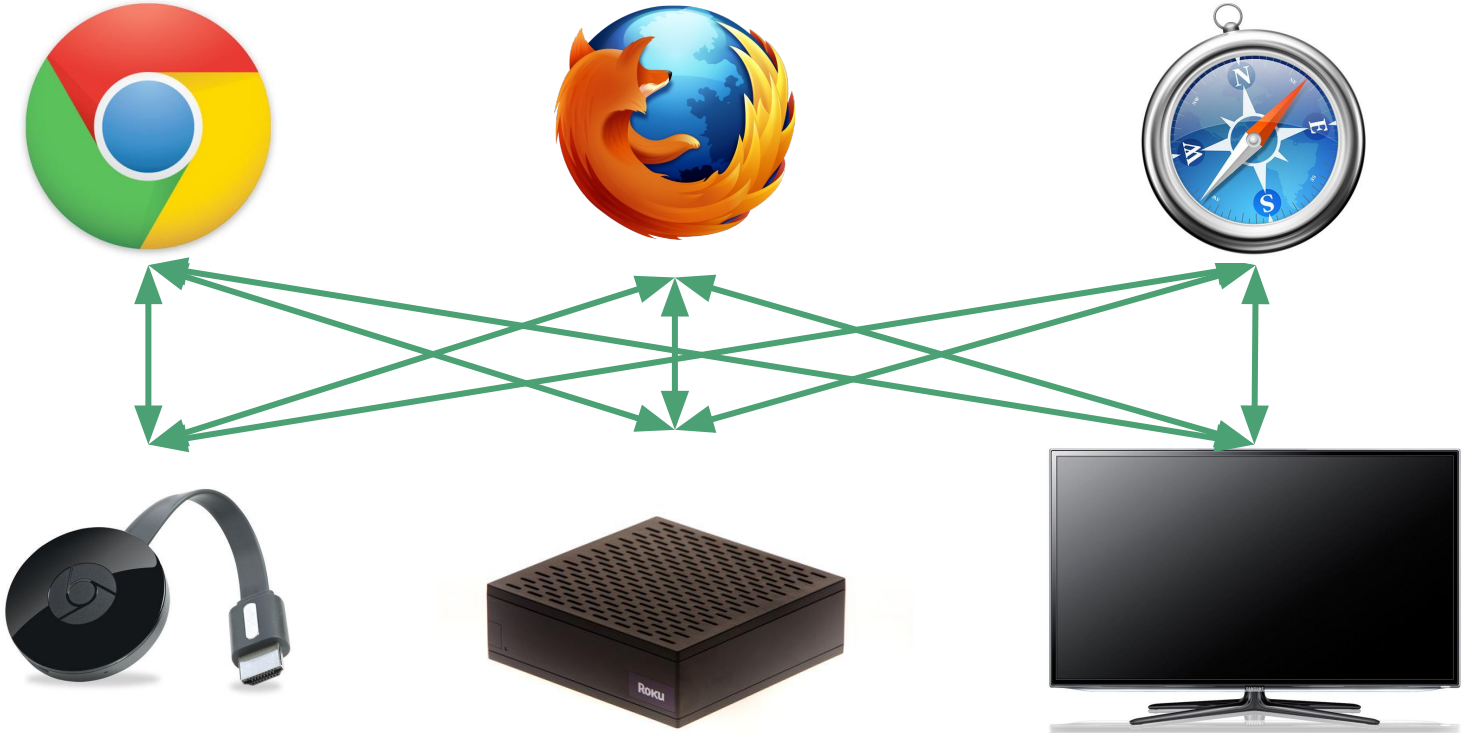
Controllers and receivers (or remote playback devices) on **same LAN**

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

Remote Playback API: Remote playback of a **src=“...”** <audio> or <video>

Consider future use cases including **streaming** and **cross-LAN** connections

Open Screen Protocol



Functional Requirements

1. Discovery of 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

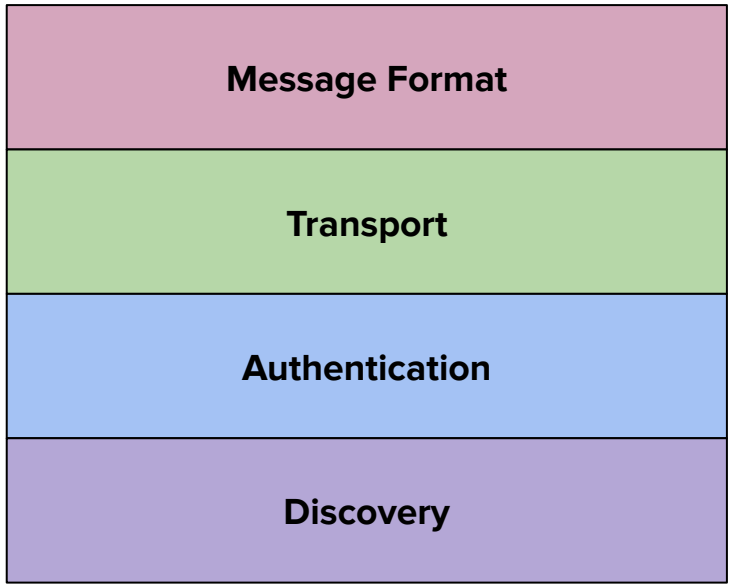
Preserving privacy and security

Resource efficiency (battery, memory)

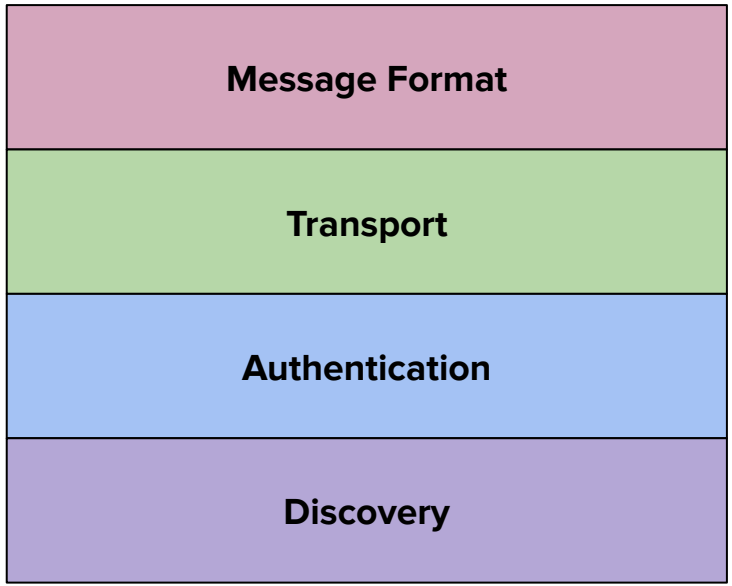
Implementation complexity on constrained devices

Extensibility and upgradeability

Open Screen Protocol - Stack



Open Screen Protocol - Current Approach



Message Format	**CBOR (RFC 7049)**
Transport	**QUIC (Draft RFC)**
Authentication	**TLS 1.3 (RFC 8446) with mutual auth**
Discovery	**mDNS (RFC 6762) / DNS-SD (RFC 6763)**

What has been accomplished

- Requirements analysis, research into alternatives
- Decision to pursue the current protocol stack
- Two authentication approaches defined:
  - Challenge/Response w/ HKDF
  - J-PAKE (backup plan in PR)
- V1 spec document has reached completion as a draft
- Open Screen Protocol Library implements part of 1.0 spec

Major work items remaining to complete 1.0 spec

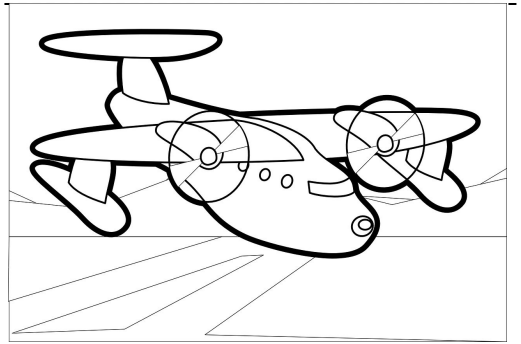
- Finalize authentication mechanism(s)
  - Consultation with crypto experts
  - TLS 1.3 implementation
  - UI guidelines
- ID issues
- Presentation API message details
- Remote Playback API message details
- Capabilities and extensions

Major work items remaining for wide review

- Remote Playback API requirements (pull request)
- TAG Explainer
- Document on custom schemes (like cast:, hbbtv:)
- Additional security analysis
- Additional items to be discussed in Day 2

Implementation: OSP Library

Landed!



Full mDNS support

Full QUIC support

Platform abstraction layer (Mac/Posix/Linux)

CBOR support via CDDL codegen

Presentation API support

Demos in C++ and Go

En route!

Final authentication implementation

Remote Playback support

Chromium integration (controller)

Forward-looking work

- **Media streaming (pull request, merged)**
  - May need add to community group scope
- **LAN traversal (ICE) support**
- Backup/alternate discovery
- WebTransport/WebCodecs polyfill implementations
- Other "V2" features

Open Screen Protocol Spec 1.0

Peter to add slides on 1.0 spec

All the spec changes since TPAC in one big diff

Some changes we had agreed on

Some changes need agreement

Some changes might need agreement, but might not

(more slides coming up for each category)

Already agreed on; now done

- mDNS with "_openscreen._udp.local"
- Minimal mDNS TXT info: Fingerprint and "timestamp"
- WIth some data exhanged over QUIC being "unverified"
- Messages serialized with (key prefix, CBOR) within QUIC streams
- Message use CBOR integer keys (and sometime positional keys)
- Considered dropping JPAKE for challenge/response
- If that doesn't work considering "JPAKE after TLS"
- Presentation messages
- Remote Playback messages
- Media time is a media-time value+scale; other timestamps are microseconds

Already agreed on; now done (part 2)

- audio-frame with position-based keys, as proposed
- video-frame is normal messages, as proposed
- Common request/response pattern
- Common error codes and types

Done, but probably needs agreement

- Comment in CDDL indicates type key for a given message (see https://github.com/webscreens/openscreenprotocol/blob/gh-pages/messages_appendix.cddl)
- mDNS instance name is a truncated display name
- receives-audio/receives-video capabilities
- No length prefix (didn't make it easier)
- Type key prefix is a QUIC uvarint (2 bits for size)
- changed-text-tracks on remote playback controls allows for adding and removing cues instead of separate method
- added-text-tracks allows for adding text tracks

Probably doesn't need agreement, but might

- Terminology of "agent" for an impl of OSP
- Terminology of "controller" and "receiver" borrowed from API specs
- Terminology of "sender" and "receiver" for media streaming
- mDNS fingerprint must be sha-512 or sha-256, not md5 or md2
- mDNS timestamp is now a version called "mv" instead of a "ts"
- If no JPAKE, HKDF params (32-byte salt, 2^15 const function, scrypt block size 8, scrypt parallelization parameter 1, fingerprint hash using sha-256)
- Presentation receiver chooses connection ID (makes scoping easier)
- HTTP headers are key/value pairs, not a big blob of HTTP/1.1
- Presentation API protocol defined independent of Presentation API, and then a separate section maps the Presentation API onto the Presentation API protocol

Probably doesn't need agreement, but might (part 2)

- enabled-audio-track-ids instead (set of IDs) of audio-tracks-enabled (bools)
- Streaming payload is "bytes" not "any"
- "frame sequence number" instead of "frame ID"
- Streaming capabilities has max audio channels and min bit rate on receive side
- Streaming capabilities has min-video-bit-rate
- Streaming capabilities has color profiles
- Streaming capabilities has native resolutions and supports-scaling

Not done

- **PAKE or not**
- **Future extensions... a capabilities mechanism to be determined. (Issue 123; PR 171)**
- **Be explicit about 0-length connection ID (issue 169; PR 170)**
- Type key allocations: 0-9 reserved, 10-63 small+frequent, 64-99 reserved, 100-1XX used in the spec; 1XX-999 reserved, 1000- available for non-standard extensions
- Remoting (v2?)
- Streaming start/stop/negotation v2
- "<source> extended mime type, media query (replace url here)"
- Support for remote playback of cues vs activeCues

Not done (Part 2)

- Issue(139): Clarify scoping/uniqueness of request IDs. (future slides)
- Issue(145): Watch ID Uniqueness (future slides)
- Issue(147): Remote playback ID uniqueness (future slides)
- Issue(107): Define suspend and resume behaviors for discovery protocol.
- Issue(108): Define suspend and resume behavior for connection protocol.
- Issue(160): Refinements to Presentation API protocol.
- Issue(159): Refinements to Remote Playback protocol
- Issue(158): Algorithm for what messages to send when local/remote media element changes.
- Specify where streaming codec names are defined (v2)
- Specify where streaming codec-specific parameters are defined. (v2)
- Add streaming capabilities for HDR

Open Screen Protocol: Authentication

Mutual Auth with Challenge/Response

In Lyon, we decided to explore the simpler (than JPAKE) challenge/response. (https://github.com/webscreens/openscreenprotocol/issues/122)

It worked out! We got clearance from Security People and specced it all out.

But...

While implementing it **we realized it would require 32MB of RAM** because, which is a lot for some devices. It turns out that in order to use low-entropy PINs, you should use a "memory hard" hashing algorithm which requires memory.

How much memory is enough? That's not clear yet. Waiting to hear back.

Mutual Auth with JPAKE

So while waiting to hear back from Security Folks about how much memory, we specced out the **back up plan, a PAKE**: https://github.com/webscreens/openscreenprotocol/pull/164

Does it also requires lots of memory? Not clear yet. Waiting to hear back.

Also, which PAKE? Security people are pushing us toward SPAKE2 or OPAQUE instead of JPAKE, but they aren't done yet. The PR above is JPAKE.

What now?

```
function chooseMutualAuthProtocol() {
  writeSpecFor(USE_CHALLENGE_RESPONSE);
  writeSpecFor(USE_JPAKE);
  const promise = securityFolks.askHowMuchMemoryWeNeed();
  // We are here
  const response = await promise;
  if (!response.challengeResponseNeedsAlot)
    return USE_CHALLENGE_RESPONSE;
  else if (!response.pakeResponseNeedsAlot)
    return await chooseAPake();
  else
    // Let's hope we don't end up here
    ???
}
```

Oh, and about those PINs

We figured out a way to determine who shows the PIN and who inputs the PIN and when.

Before auth, each side sends the other side the following message:

```
auth-capabilities = {
 0: uint ; psk-ease-of-input, 0-100 (0 is impossible, 100 is easy)
 1: [* &(numeric: 0, alphanumeric: 1, qr-code: 2)] ; psk-input-methods
}
```

The side that can enter the PIN more easily enters it.
The other side shows it.
The entering side indicates which types it enter.

(In a tie, the server presents and the client inputs)

Oh, and about those PINs (examples)

Phone: "ease: 75; inputs: numeric, alphanumeric, QR code"
TV: "ease: 5; inputs: numeric"
Result: TV shows number or QR code or both; phone enters one

Phone: "ease: 75; inputs: numeric, alphanumeric, QR code"
Speaker: "ease: 0; inputs: numeric"
Result: speaker "presents" a number; phone enters one

TV: "ease: 5; inputs: numeric"
Speaker: "ease: 0; inputs: numeric"
Result: speaker "presents" a number; TV enters one with remote

Decision Making

Do we have consensus on this much?

```
auth-capabilities = {
 0: uint ; psk-ease-of-input, 0-100 (0 is impossible, 100 is easy)
 1: [* &(numeric: 0, alphanumeric: 1, qr-code: 2)] ; psk-input-methods
}
```

Other things

- How to agree on PIN length? (Issue #111)
- What kinds of certificates should be exchanged? (Issue #135)
- How to show trusted and untrusted data? (Issue #118)

(One slide each coming up)

How to agree on PIN length?(Issue #111)

The amount of entropy you need is related to how hard the hashing function you have is. Still waiting to hear back from Security People on that. Hopefully 20-40 bits of entropy is enough.

But what if 2 OSP agents pick different lengths?
- What if 20 is OK for you but I want to require 40?
- What if someone says they only want 1 bit? Or 100?

Proposal:

1. Add a "psk-min-bits-of-entropy" to auth-capabilities.
2. Make the min value of psk-min-bits-of-entropy 20 bits.
3. Make the max value of psk-min-bits-of-entropy 60 bits.
4. If unset, the default min is 20 bits.

That way, everyone will have a reasonable value, but some flexibility in choosing a min.

Decision Making

Do we have consensus on this mechanism?

`psk-min-bits-of-entropy`

What kinds of certificates exchanged? (Issue #135)

Questions:

1. EC or RSA certs?
2. What certificate extensions?
3. What TLS 1.3 extensions?

Proposals:

1. Require acceptance of EC certs, so everyone can generate EC certs (But you can still use RSA if you want)
2. Certificate extensions should be ignored unless otherwise stated
3. Not require or prohibit TLS 1.3 extensions, but look for implementation feedback on this

Decision Making

Do we have consensus on this?

Require acceptance of EC certs

How to show trusted and untrusted data? (Issue #118)

What do we show the users about auth? Things like:

1. When a remote agent is requesting auth, such that a PIN or PIN entry pops up on your TV.
2. The auth state of the remote agent (unauthenticated, authenticated, failed to authenticate), such as in a list of devices on the network
3. When data about an agent is unauthenticated (such as the name or icon of an unauthenticated device in a list of devices)
4. When a remote agent is authorized but changes its name or icon or how it looks in some other way.

Decision Making

What kind of text are we looking for?

Implementers should consider the following when designing UX showing information about remote agents:

- Whether the remote agent is authenticated or not
- Whether the remote agent has previously failed authentication or not
- Whether the display name and icon of the remote agent has changed since it was authenticated.

Open Screen Protocol: Security & Privacy

Things to talk about

Some open security and privacy issues (outside of authentication)

- Securing TLS 1.3 implementations (Issue #130)
- Mitigations against remote network attackers (Issue #131)
- Notifying endpoints when a new connection is created (Issue #143)
- Attestation (endpoint-to-endpoint trust) (Issue #114)

Securing TLS 1.3

Issue: Like any authentication protocol, TLS 1.3 is subject to various kinds of attack methods.

1. Downgrade attacks
2. Weak ciphers
3. Timing/side channel attacks
4. Key compromise
5. Replay attacks with 0-RTT ("early") data

Ref: draft-camwinget-tls-use-cases, RFC 8446 C-E

Securing TLS 1.3

What are some protections we can provide in the spec?

Downgrade attacks	Forbid < TLS 1.3
Weak ciphers	Require strong ciphers
Timing/side channel attacks	Only allow constant-time (AEAD) ciphers
Key compromise	Requirements around pre-shared keys
Replay attacks with 0-RTT data	Allow early data for some messages

Securing TLS 1.3

What are next steps?

Forbid < TLS 1.3	Update spec
Require strong ciphers	Research alternatives, make proposal
Only allow constant-time (AEAD) ciphers	Update spec
Requirements around pre-shared keys	Research alternatives, make proposal
Allow early data for some messages	Make proposal (vs banning early data)

Expert review of final implementation of TLS 1.3 is important.

Decision Making: TLS 1.3

Do we have consensus on the following?

Forbid < TLS 1.3	**Update spec**
Require strong ciphers	Research alternatives, make proposal
Only allow constant-time (AEAD) ciphers	**Update spec**
Requirements around pre-shared keys	Research alternatives, make proposal
Allow early data for some messages	**Make proposal (vs banning early data)**

Mitigating against remote network attackers

Issue: Endpoints outside the LAN will be able to reach Open Screen agents.

Simplest: Do nothing. (Mutual authentication should prevent any misuse.)

Detect: Tell the user when external connection attempts are observed.

Mitigate: Require extra data in the ClientHello only found through mDNS (like fp)

Restrict: Ban connections from non-RFC1918 IP addresses.

Mitigating against remote network attackers

Issue: Devices outside the LAN may be able to reach Open Screen agents.

Simplest: Do nothing. (Mutual authentication should prevent any misuse.)

Recommend: Tell the user when ~~external~~ failed connection attempts are observed.

~~**Mitigate:** Require extra data in the ClientHello only found through mDNS (like fp)~~

~~**Restrict:** Ban connections from non-RFC1918 IP addresses.~~

Decision Making: Remote network attackers

Issue: Devices outside the LAN may be able to reach Open Screen agents.

Decision:

Include the following recommendation in the spec:

Tell the user when failed connection attempts are observed.

Endpoint notification

Issue: Presentation API recommends that receivers notify controllers of new connections, to let users know they may be exchanging data with other parties.

Endpoint notification

Issue: Presentation API recommends that receivers notify controllers of new connections, to let users know they may be exchanging data with other parties.

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

presentation-connection-open-response = {
  response
  1: &result ; result
  2: uint ; connection-id
  3: uint ; num-open-connections
}
```

+

Whenever a connection is opened or closed, the receiver sends to all other controllers with an open connection:

```
presentation-connection-change-event = {
  1: text ; presentation-id
  2: uint ; connection-count
}
```

Decision Making: Endpoint notification

Issue: Presentation API recommends that receivers notify controllers of new connections, to let users know they may be exchanging data with other parties.

Decision: Add the following messages to be sent by the receiver when a connection is opened.

```
presentation-connection-open-response = {
  response
  1: &result ; result
  2: uint ; connection-id
  3: uint ; num-open-connections
}
```

```
presentation-connection-change-event = {
  1: text ; presentation-id
  2: uint ; connection-count
}
```

Attestation: endpoint-to-endpoint trust

Mutual authentication prevents a network MITM and provides a unique and stable identifier (fp).

It does not prove anything about the origin or attributes of each endpoint (manufacturer, model, software/OS, etc.)

An attestation protocol would allow one endpoint to verify information about another endpoint based on trust by some other party.

Attestation: endpoint-to-endpoint trust

Questions:

1. What facts do implementations care about verifying? Who provides them?

2. What are acceptable mechanisms to verify third-party trust?

    - User verification

    - Certificates and signing

3. Who does the verification? User agents, applications, other?

4. Who needs the verified information?

Decision Making: Attestation.

No decision yet. Gather requirements around use cases for attestation from inside and outside the group.

1. What facts do implementations care about verifying? Who provides them?

2. What are acceptable mechanisms to verify third-party trust?

    - User verification

    - Certificates and signing

3. Who does the verification? User agents, applications, other?

4. Who needs the verified information?

General Protocol Issues

Messages and IDs

Each CBOR message has a numeric type identifier used to prefix it in the QUIC stream. Each key in each message also has its own numeric ID.

Agent capabilities are enum values

Many messages are paired into requests and responses. Each request has a numeric identifier.

A request that can have multiple responses creates a "watch" with its own identifier.

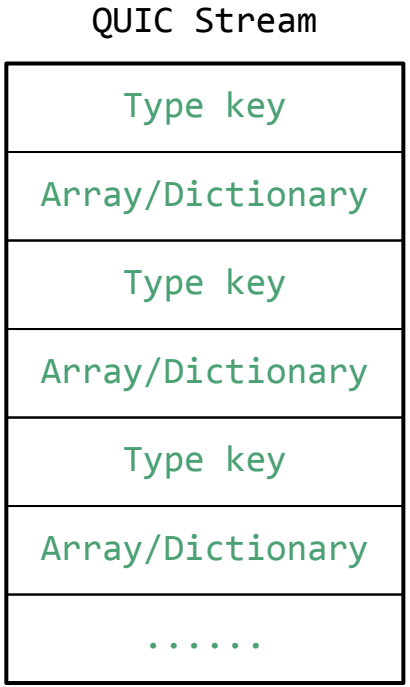
Question: How do we assign these identifiers?

Things to talk about

Some open questions about various IDs we use.

- How to assign message type keys
- How to assign capabilities enum values
- How to assign request IDs (Issue #139)
- How to assign watch IDs (Issue #143)
- ~~Do we want CBOR semantic tags?~~

Message Type IDs



Length of type key depends on the value.

Range	# Bytes
0-63	1
64 to 16,383	2
16,384 to 2^{30}-1	4
2^{30} to 2^{62}-1	8

Message Type IDs

Use shorter types for more frequent messages, and group related messages.

Range	Message types
1 to 48	`audio-frame, video-frame, presentation-connection-message, ...`
64 to 8,192	`agent*, auth*, remote-playback-state, remote-playback-modify* presentation-*, remote-playback-*`
16,384 to 2^{29}-1	`Reserved`
>= 2^{30}	`Reserved`

Range	Message types
49-63	`For non-standard extensions`
8193-16383	`For non-standard extensions`
2^{29}-(2^{30}-1)	`For non-standard extensions`

Capabilities

```
agent-capability = &(
 receive-audio: 1
 receive-video: 2
 receive-presentation: 2
 control-presentation: 3
 receive-remote-playback: 4
 control-remote-playback: 5
 receive-streaming: 6
 send-streaming: 7
)
```

The sizes here aren't too important. We could just reserve a range ...

Additional notes on message type IDs

- Length prefixing was just removed, since it made implementations more complicated.
- The spec doesn't mention how to assign type ids to new messages, or which type ids are for extensions. We can write up the previous table in an appendix.
- Do we want an IANA registry for external (extensions) type keys?
- We also have numeric IDs for the keys in each message. We can allow agents to extend these as well.
- Agents can use a large number or a string for extended message keys.

Request and Response IDs.

For an agent to know what message is a response to a request, each has an ID.

```
// Controller => Receiver
presentation-connection-open-request = {
  request => { 0: uint ; request-id }
  1: presentation-id ; presentation-id
  2: connection-id ; connection-id
}

// Receiver -> Controller
presentation-connection-open-response = {
  response => { 0: uint ; request-id }
  1: &result ; result
}
```

We should avoid collisions across QUIC connections between two agents.

Request and Response IDs: Option 1

Each agent keeps track of the next request ID to use for an endpoint, but is allowed to forget.

Each agent starts with 1 and increments by 1 for each message.

Controller (id 0x4567)

Endpoint fp	Next Request ID
0x1234	0005

Receiver (id 0x1234)

Endpoint fp	Next Request ID
0x4567	0006

If an agent sees a request with ID 1, then it discards all previous requests in-flight.

Request and Response IDs: Option 2

If you forget, tell the other side:

```
reset-agent-state-request = {
  request
}
```

The other side acts as though it just connected to you (but authenticated). It drops all watches and pending requests. You can't trust any responses or watched events until you get a response to this message.

Problem: if I forget everything, I don't remember to tell you to reset. But we could make the rule "if you don't have remember any state, you must send this", so this would be sent on all first connections between two agents, which might be confusing.

Request and Response IDs: Option 3

Amnesia detection token:

```
agent-info = {
 ? 5 : uint ; state-token
}
```

1. Initially a random number
2. In future reconnects, reuse if you remember state
3. If the remote side changes, assume it was reset

Watch IDs

Watch IDs are like request IDs that can have multiple responses.

We can use the same tables to allocate Watch IDs.

Endpoint fp 0x4567

Endpoint fp	Next Request/Watch ID
0x1234	0005

Endpoint fp 0x1234

Endpoint fp	Next Request/Watch ID
0x4567	0006

Proposal: use the same solution we picked for request IDs for watch IDs.

Presentation API Protocol Issues

Things to talk about

Some open questions about Presentation API protocol messages, especially around close/terminate.

- Do we need `presentation-connection-close-response`? (Issue #138)
- How should presentation connections handle not getting it? (Issue #137)
- Is a presentation close/terminate a response or an event? (Issue #124)

Closing a connection from the controller

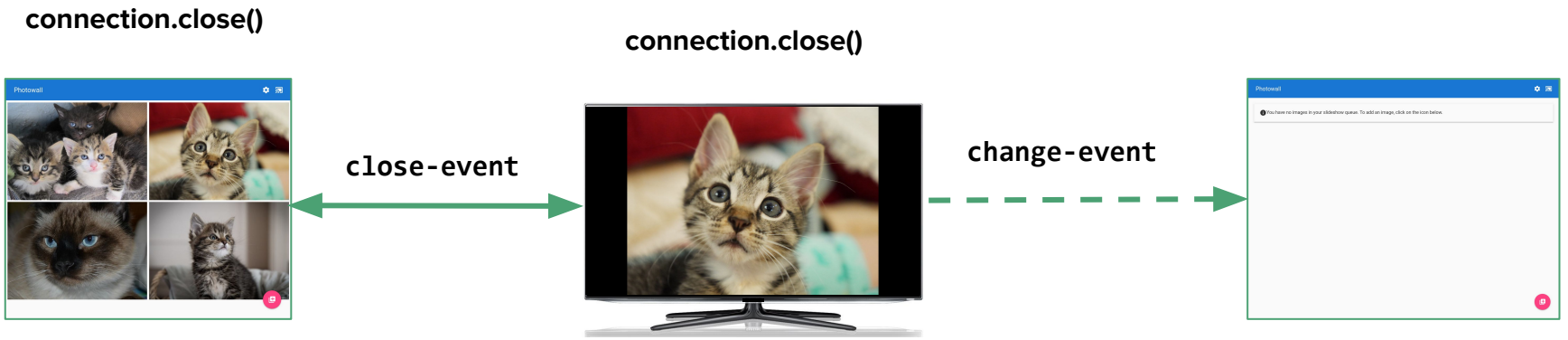
Other connections are not notified.

Closing a connection from the receiver

The controller whose connection was closed is notified.

Proposal: Always use an event.

Just need to tell JS why. Agent doesn't have to wait for a response.



Receiver also sends a `presentation-connection-change-event` to other controllers.

Can remove `close-request` and `close-response` from the spec.

Proposal: This works for termination too.

Agent doesn't have to wait for a response. Just need to tell everyone.

Receiver also sends a `presentation-termination-event` to other controllers.

Can remove `termination-request` and `termination-response` from the spec.

Remote Playback API Protocol Issues

Things to talk about

Some open questions about Remote Playback API.

- Review the Remote Playback API requirements (Issue #13, pull request)
- Disconnect/Reconnect of Remote Playback API (Issue #89)
- Multiple Controllers of Remote Playback (Issue #124)
- Assigning Remote Playback ID (Issue #137)

Remote Playback Protocol Requirements (1/2)

Controlling UA must be able to:

- Find out whether there is at least one compatible remote playback device available for a given HTMLMediaElement, both instantaneously and continuously.
- Initiate remote playback of an HTMLMediaElement to a compatible remote playback device.
- Send media URL and text tracks from an HTMLMediaElement to a compatible remote playback device.

Remote Playback Protocol Requirements (2/2)

- During remote playback, the controlling user agent and the remote playback device must be able to synchronize the media element state of the HTMLMediaElement.

- During remote playback, either the controlling user agent or the remote playback device must be able to disconnect from the other party.

- The controlling user agent should be able to pass locale and text direction information to the remote playback device to assist in rendering text during remote playback.

Remote Playback: Disconnecting and Reconnecting

Remote playbacks have a `remote-playback-id` assigned by the controller.

We could expose this to JS:

```
interface RemotePlayback : EventTarget {
    optional long remotePlaybackId;
    Promise<void> reconnect(long remotePlaybackId);
}

video.remote.reconnect(12345).then(() => video.play());
```

What does it mean for **video** when it reconnects?

Remote Playback: Multiple Controllers

The same idea could be used to allow multiple controllers for a remote playback.

```
interface RemotePlayback : EventTarget {
    optional long remotePlaybackId;
    Promise<void> reconnect(long remotePlaybackId);
}

const video = createElement('video');
video.remote.reconnect(12345).then(() => video.play());
```

How does the page get permission to control the playback?

The Synced Media in Presentation API proposal offers one approach.

Remote Playback IDs: Single Controllers

Remote Playback IDs are like request IDs.

They are created by an agent when a remote playback starts.

Endpoint fp 0x4567

Endpoint fp	Next ID
0x1234	0005

Endpoint fp 0x1234

Endpoint fp	Next ID
0x4567	0006

For a single controller, we can just use the same solution as request IDs and watch IDs.

All of these IDs **must be scoped by endpoint fp** to avoid leaking permissions.

Remote Playback IDs: Multiple Controllers

To support multiple controllers, we need to have IDs that are unique across all endpoints and connections.

This would mean generating GUIDs like we do for Presentation IDs.

Decision: Remote Playback IDs

Proposal: Use a strong unique ID (like a Presentation ID) to allow future use cases with multiple controllers.

Open Screen Protocol
Day 2

Peter Thatcher (pthatcher@google.com)
Mark Foltz (mfoltz@google.com)
TPAC 2018

Day 2 - Outline

Remaining Day 1 Topics

Streaming & Capabilities

Open Screen Library Update

Presentation API Features

Planning

(If time) Other New OSP Features

Streaming and Capabilities

Streaming basics

Server sends to receiver:

audio-frame
video-frame

As demonstrated at TPAC

A frame references an *encoding* which contains typically redundant information (codec, time scale)

Starting a streaming session

1. Sender offers various encodings
2. Receiver selects encodings (based on codec)
3. Receiver specifies other parameters like target resolution, max framerate
4. Sender sends to receiver
5. Receiver renders with minimal buffering and no history

Can modify the selected encodings and parameters any time.

During a streaming session

- Receiver can ask sender for a key frame
- Receiver and sender can send their peer stats
- We need to figure out what stats (we put in some basic ones)

Remoting

Like streaming, but forwarding (only transcoding as necessary) and buffer w/history. Here's how:

1. Receiver sends sender capabilities (just like what was presented at TPAC)
2. Sender sends info about encodings
3. Sender sends media to receiver

Basically, extends remote playback protocol with part of the streaming protocol.

What about text/data

At TPAC and in the original PRs for streaming, I included it. But in the latest PR, I removed it. It just felt to abstract, generic, and without a use case.

Streaming doesn't have a need for text/data.

Remoting does, but it can use the remote playback text track adding that's already there.

So what use case is there? Perhaps touch screen inputs, but that's data going in the opposite direction, so we might want to do it differently.

I recommend we wait until we have a concrete use case and then add it later.

Open Screen Protocol Library

Open Screen Library Update

- 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; platform abstraction layer
3. Small footprint (binary and memory size); YAGNI principle
4. Full OSP implementation
5. Embedder is responsible for rendering HTML5 and media
6. Extensible with new features / protocols

OSP Library: What can it do right now?

Advertise an Open Screen agent and listen for Open Screen agents with mDNS.

Obtain an agent's friendly name, port and IP address.

Connect to an Open Screen agent with QUIC.

Send and receive `presentation-*` messages.

All exercised by the demo in `osp/demo`.

Open Screen Library: New since 10/18

Features and protocol support

- Full CBOR code generation from CDDL
- Full QUIC support (client and server)
- Implement PresentationController, PresentationReceiver, PresentationConnection
- Support for reading/writing mDNS TXT data.
- Implementation of presentation spec in Go
- Demo with presentation controller, receiver, connection, messages

Open Screen Library: New since 10/18

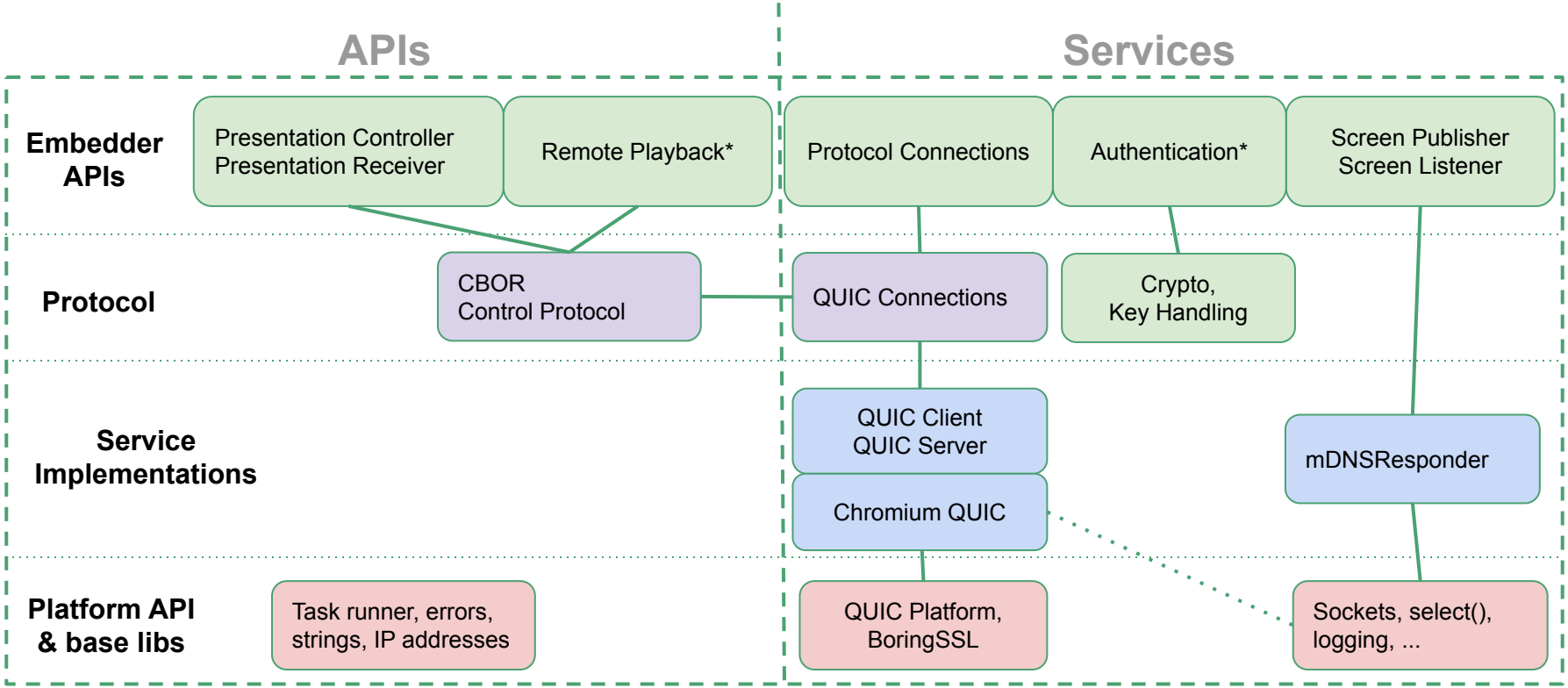
Platform and CI infrastructure

- absl:: support including optional and string_view
- IPv6 multicast support.
- Full Mac platform support including Mac sockets and Mac builders
- Support for clang as well as gcc with CI. ASAN support in builders
- std::chrono support for time
- Task runner (embedder provided thread)
- Lots of work to align dependencies with Chromium

OSP Library: High Level Structure

`osp/public`	Start/stop mDNS and QUIC. Or provide your own implementation of them. Discovery OSP agents. Connect/disconnect from OSP agents.
`osp/public/presentation`	Interfaces to use Presentation protocol.
`osp/impl` `osp/msgs`	Implements the APIs in `osp/public`. CBOR reading/writing from CDDL.
`platform/api` `platform/base`	Platform abstraction definition Platform utilities
`platform/{linux,mac,posix}`	Provided implementations of `platform/api`.
`osp_base/`	Basic utilities

Open Screen Library: Architecture



* To be implemented

Open Screen Library: Embedder APIs

```
class openscreen::presentation::Controller {
  ReceiverWatch RegisterReceiverWatch(const std::string& url, ReceiverObserver* observer);

  ConnectRequest StartPresentation(const std::string& url, const std::string& service_id,
                                   RequestDelegate* delegate, Connection::Delegate* conn_delegate);

  ConnectRequest ReconnectPresentation(const std::string& presentation_id, const std::string& service_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: Bringing your own services

```
auto mdns_listener =
    MdnsServiceListenerFactory::Create(listener_config, &listener_observer);
auto* network_service = NetworkServiceManager::Create(
    std::move(mdns_listener), nullptr, std::move(connection_client), nullptr);
auto controller =
    std::make_unique<presentation::Controller>(platform::Clock::now);
network_service->GetMdnsServiceListener()->Start();
network_service->GetProtocolConnectionClient()->Start();
```

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

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

Open Screen Library: Code generation from CDDL

```
; type key 14
presentation-url-availability-request = {
  request
  1: [* text] ; urls
  2: microseconds ; watch-duration
  3: uint ; watch-id
}

; type key 15
presentation-url-availability-response = {
  response
  1: [* url-availability] ; url-availabilities
}
```

Open Screen Library: Code generation from CDDL

CDDL

```
; type key 14
presentation-url-availability-request = {
  request
  1: [* text] ; urls
  2: microseconds ; watch-duration
  3: uint ; watch-id
}
```

C++

```
struct PresentationUrlAvailabilityRequest {
  uint64_t request_id;
  std::vector<std::string> urls;
  uint64_t watch_duration;
  uint64_t watch_id;
};

ssize_t EncodePresentationUrlAvailabilityRequest(
    const PresentationUrlAvailabilityRequest& data,
    uint8_t* buffer,
    size_t length);
ssize_t DecodePresentationUrlAvailabilityRequest(
    const uint8_t* buffer,
    size_t length,
    PresentationUrlAvailabilityRequest* data);
```

Open Screen Library: Platform API

UDP Socket

Socket Event Waiter

Logging

Network Interface

Task Runner

Time

Implementations provided for linux, Mac, posix

```
class UdpSocket {

static ErrorOr<UdpSocketUniquePtr> Create(
    Version version);

Error Bind(const IPEndpoint& local_endpoint);

Error SetMulticastOutboundInterface(
    NetworkInterfaceIndex ifindex);

Error JoinMulticastGroup(
    const IPAddress& address,
    NetworkInterfaceIndex ifindex);
```

Open Screen Library: Platform API

```
class UdpSocket {

static ErrorOr<UdpSocketUniquePtr> Create(
    Version version);

Error Bind(const IPEndpoint& local_endpoint);

Error SetMulticastOutboundInterface(
    NetworkInterfaceIndex ifindex);

Error JoinMulticastGroup(
    const IPAddress& address,
    NetworkInterfaceIndex ifindex);
```

```
ErrorOr<size_t> ReceiveMessage(void* data,
    size_t length,
    IPEndpoint* src,
    IPEndpoint* original_destination);

Error SendMessage(const void* data,
                  size_t length,
                  const IPEndpoint& dest);
```

OSP Library: What is next?

Implement authentication messages, protocol, and TLS support.

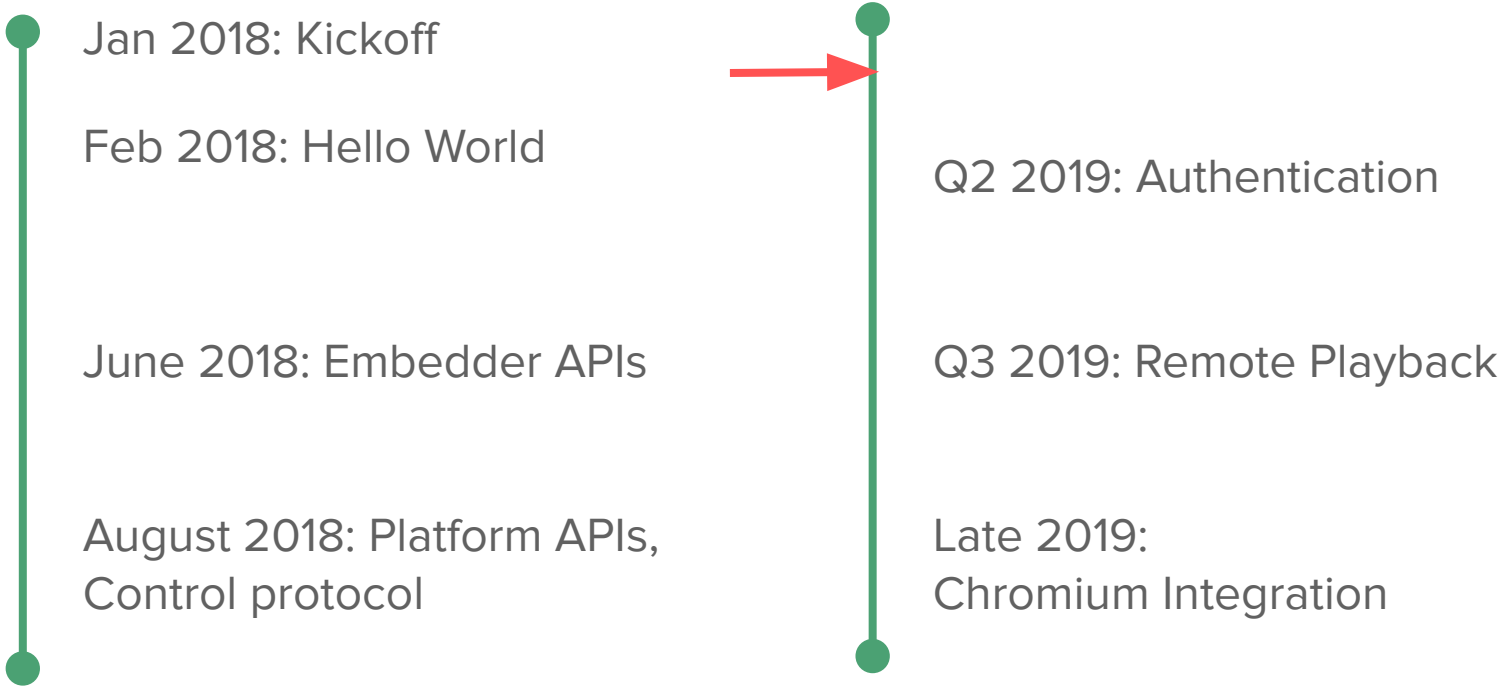
Integration with embedder provided task runner (versus manual poll-looping).

Add Remote Playback messages and APIs.

Add streaming messages and APIs.

Chromium integration.

Open Screen Library: Timeline



Open Screen Library: Continuous Integration

Have builders for Linux, Mac, gcc and clang.

Trybots/commit queue enabled (run tests before push).

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

Presentation API Features

Local Presentation Mode

Local Presentation Mode

Explainer

Add dictionary

Presentation and Remote Playback APIs Proposals for enhancements

Takumi Fujimoto
takumif@google.com

3 proposals

Make Presentation and Remote Playback APIs work better together:

1. One method to invoke both APIs (one button and on dialog)
2. Synced media in Presentation API

Supporting more device types:

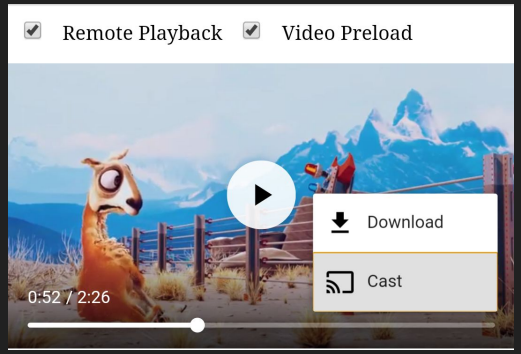
3. MSE streaming in Remote Playback API

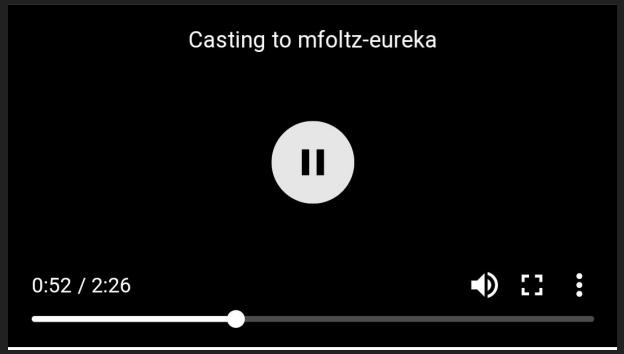
Recap: Presentation API

1. Controlling web page requests presentation of URL
2. Browser lists devices compatible with URL; user selects one
3. Browser request that receiver presents URL; receiver does so
4. Controlling and receiving pages exchange messages
5. Either side closes the connection or terminates the presentation.

Recap: Remote Playback API

1. Controlling web page requests remote playback of <audio> or <video>
2. Browser lists devices compatible with element; user selects one
3. Browser request that receiver plays; receiver does so
4. Media commands are forwarded to the remote playback device and media state is sent back to the browser





Proposal 1: One method to invoke both APIs

State of the current APIs

Two separate methods to start sessions, `PresentationRequest.start()` and `RemotePlayback.prompt()`

Each shows a potentially different list of receiver devices to choose from, so user may need to open two different device selection dialogs to find a device

Proposal 1

Let `PresentationRequest` start either a presentation or remote playback by giving it an optional `sourceMedia` attribute

The UA can choose to either start a presentation (using the presentation URL) or remote playback (using the source media), depending on the device chosen

If remote playback is chosen, `PresentationRequest.start()` is resolved with `null`, and `sourceMedia.remote` is updated and has events fired accordingly

WebIDL

```
partial interface PresentationRequest {

  Promise<PresentationConnection?> start();

  attribute HTMLMediaElement? sourceMedia;

};
```

WebIDL (existing API)

```
partial interface RemotePlayback {

  readonly attribute RemotePlaybackState state;

           attribute EventHandler        onconnecting;

           attribute EventHandler        onconnect;

           attribute EventHandler        ondisconnect;

};
```

Sample code (sender page)

```
<video id="my-video" src="https://example.com/file.mp4"></video>

<script>
  const request = new PresentationRequest('https://example.com/receiver.html');
  request.sourceMedia = document.querySelector('#my-video');

  request.start().then(connection => {
    if (connection) {
      // Presentation has been started.
    } else {
      // Remote playback has been started.
      // request.sourceMedia.remote.state is now 'connected'.
    }
  });
</script>
```

Sample code (sender page)

```
let remoteDevice = await SecondScreen.prompt(
  ["presentation-receiver", "receives-audio", "receives-video"]
  ["remote-playback-receiver", "receives-audio"]
]]);
if (remoteDevice.supports("presentation-receiver")) {
  let request = new PresentationRequest(remoteDevice, 'https://example.com/');
  request.start();
} else if (remoteDevice.supports("receives-video")) {
  document.querySelector('#my-video').remote.start(remoteDevice);
} else {
  document.querySelector('#my-audio').remote.start(remoteDevice);
}
```

Sample code (sender page)

```
let preso = new PresentationRequest(['example.com/myvideo.html', ...])
let remote = document.querySelector('#my-video').remote;
let chosen = await SecondScreen.prompt([preso, remote]);
if (chosen == preso) {
  // as if you called preso.start()
} else if (chosen == remote) {
  // as if you called remote.prompt()
}
```

Proposal 2: Synced media in Presentation API

Limitations of the current APIs

Presentation API

- Website needs to define custom messages for controlling media playback (Or rely on the Cast SDK or Shaka Player, which don't interop)

Remote Playback API

- No stylized or custom UI elements (controls, subtitles, ads, splash screen)
- No MSE/EME support (unless streamed/mirrored via the sender)

The APIs cannot be used together, so the developer must pick one

Proposal 2

Automatically sync media playback in Presentation API sessions by allowing the sender and receiver pages to designate controlling and controlled `HTMLMediaElements`

While elements on both sides are set, their playback is synced, like with Remote Playback

Their `HTMLMediaElement.remote` attributes are updated and have events fired accordingly

WebIDL

Controller side:

```
partial interface PresentationConnection {
  attribute HTMLMediaElement? controllingMedia;
};
```

Receiver side:

```
partial interface PresentationReceiver {
  attribute HTMLMediaElement? controlledMedia;
};
```

Sample code (controller page)

```
<video id="sender-video" src="https://example.com/file.mp4"></video>

<script>
  const request = new PresentationRequest('https://example.com/receiver.html');
  const connection = await request.start();
  const videoElement = document.querySelector('#sender-video');

  videoElement.remote.onconnected = () => {
    // Pausing the controller-side element now pauses the receiver-side
    // element as well.
    videoElement.pause();
  };

  connection.controllingMedia = videoElement;
  // |videoElement.remote.state| is now 'connecting'.
  // Once |controlledMedia| on the receiver side is set, it becomes 'connected'.
</script>
```

Sample code (receiver page)

```
<video id="receiver-video" src="https://example.com/file.mp4"></video>

<script>
  const remoteVideoElement = document.querySelector('#receiver-video');

  remoteVideoElement.remote.onconnected = () => {
    console.log('connected, playback sync is on.');
  };

  navigator.presentation.receiver.controlledMedia = remoteVideoElement;
  // |remoteVideoElement.remote.state| is now 'connecting'.
  // Once |controllingMedia| on the controller side is set, it becomes
  // 'connected'.
</script>
```

Draft explainer doc

https://github.com/takumif/presentation_api_synced_media/blob/master/explainer.md

Proposal 3: MSE streaming in Remote Playback API

State of the current Remote Playback API

Given a media element with an MSE source and/or a streaming-only receiver, the sender UA may use media remoting (forwarding of media without transcoding) to send the stream directly to the receiver

However, since the web page doesn’t know the receiver’s (or the local network’s) capabilities, the MSE stream may not be suitable for direct playback on the receiver, resulting in choppy or failed playback. The sender UA implementation can transcode the stream for the receiver, but this is expensive and can degrade the stream quality

Proposal 3

Allow the web page to observe the capabilities of the receiver and/or the local network, and alter its MSE input accordingly

Add a `capabilities` attribute to the `RemotePlayback` interface, which is set during remote playback sessions

WebIDL

```
partial interface RemotePlayback {
  Promise<RemotingRequest> offerRemoting(RemotingOffer);
  attribute eventhandler onremotingrequestchanged;

  // Option A: low-level with more info
  readonly attribute TimeRanges bufferedRanges;
  attribute eventhandler onbufferedrangeschanged;

  // Option B: less info but easier to use
  readonly attribute double remotingBitrate;

  // Option C:
  // HAVE_INSUFFICIENT, HAVE_ENOUGH_DATA, HAVE_FULL
  readonly attribute RemoteReadyState readyState;
  attribute eventhandler onreadystatechange;
};
```

WebIDL

```
dictionary RemotingOffer {
  sequence<AudioEncodingOffer> audio;
  sequence<VideoEncodingOffer> video;
}

dictionary AudioEncodingOffer {
  DOMString encodingId;
  DOMString mimeType;
}

dictionary VideoEncodingOffer {
  DOMString encodingId;
  DOMString mimeType;
  VideoResolution resolution;
  double framerate;
}
```

```
interface RemotingRequest {
  readonly attribute sequence<AudioEncodingRequest>
audio;
  readonly attribute sequence<VideoEncodingRequest>
video;
}

interface AudioEncodingRequest {
  readonly attribute DOMString encodingId;
}

interface VideoEncodingRequest {
  readonly attribute DOMString encodingId;
  VideoResolution targetResolution;
  double maxFramerate;
}
```

WebIDL

```
partial interface RemotePlayback {
  readonly attribute RemotePlaybackCapabilities? Capabilities;
  attribute EventHandler oncapabilitieschanged;
};
interface RemotePlaybackCapabilities {
  TBD
};
```

Other new OSP Features

WebTransport/WebCodecs polyfill

WebTransport

An abstract transport of streams and datagrams, implementations defined for client/server QUIC and p2p ICE+QUIC. What about OSP?

We could do something like this:

```
// Shows a dialog with video receivers capable of presentation, remote playback, or streaming
let transport = await OpenScreenTransport.prompt([
  ["presentation-receiver", "receives-audio", "receives-video"]
  ["remote-playback-receiver", "receives-audio", "receives-video"],
  ["streaming-receiver", "receives-audio", "receives-video"]]);
let message = ...;
transport.createSendStream().write(message);
```

And the app can implement all the protocol messages (polyfill). The browser just needs to implement mDNS, QUIC, and auth.

WebCodecs

An API for doing audio and video encode and decode.

OpenScreenTransport + WebCodecs allows polyfill of OSP streaming.

Backup/alternate discovery

Cloud discovery (w/o polyfill)

Receiver speaks to cloud; reachable from clients that speak to cloud

Browser speaks to cloud; shows receiver in dialog

Browser connects to receiver through cloud (perhaps bootstrapping to LAN conn)

Browser uses OSP for Presentations, Remote Playback, and Streaming messages but not for mDNS, QUIC.

Cloud discovery (w/ OpenScreenTransport + polyfill)

Receiver speaks to cloud; reachable from clients that speak to cloud

Browser speaks to cloud; shows receiver in dialog

Browser connects to receiver through cloud (perhaps bootstrapping to LAN conn)

Web app uses OSP for Presentations, Remote Playback, and Streaming messages but not for mDNS, QUIC.

Cloud discovery (w/ QuicTransport + polyfill)

Receiver speaks to cloud; reachable from clients that speak to cloud

Web app speaks to cloud; shows receiver in dialog

Web app connects to receiver through cloud (perhaps bootstrapping to LAN conn)

Web app uses OSP for Presentations, Remote Playback, and Streaming messages but not for mDNS, QUIC.

No browser support needed! Can do this today! Although the dialog is in the web page, not in the browser UI.