Talking to ChromeCasts

Perth Linux User Group, September 2016

James Henstridge <james@jamesh.id.au>


chrome

Device Overview

Control a TV from Chrome browser, Android, or iOS

Primarily marketed as a media streaming system

Hardware

Marvel Armrda 1500 Mini
- ARM Cortex A7 dual core
- HDMI 1080p60 output
- OpenGL ES 2.0 capable GPU

Memory: 512 MB

Storage: 2 GB (Gen 1), 256 MB (Gen 2)

Wifi 802.11 b/g/n (Gen 1), b/g/n/ac (Gen 2)

Software Stack

Linux based firmware

Chrome browser used to run HTML5 receiver app

RPC Message bus for communication with sender app

Reverse Engineering the RPC

Not done by me

Lots of details leaked in the Chromium source tree

Device Discovery

Via mDNS (aka. Bonjour, Zeroconf, etc)

```
$ avahi-browse -tr _googlecast._tcp
+   eth0 IPv4 Upstairs-Chromecast                          _googlecast._tcp     local
=   eth0 IPv4 Upstairs-Chromecast                          _googlecast._tcp     local
   hostname = [Upstairs-Chromecast.local]
   address = [192.168.0.52]
   port = [8009]
   txt = ["rs=" "bs=FA8FCA890C6C" "st=0" "ca=4101" "fn=Upstairs-Chromecast"
"ic=/setup/icon.png" "md=Chromecast" "ve=05" "rm=BF8BF7DFB513AE69"
"id=e393da5086494695baabe62713a0ee57"]
```

Connecting to the RPC Bus

Create TCP connection to port reported by mDNS

Establish TLS session

- Chromecast uses a unique self signed certificate
- Certificate regenerated at regular intervals

Messages are sent in either direction as length prefixed byte sequences:

4 bytes	N bytes
Length (BE)	Protobuf

Protobuf

```
message CastMessage {
  enum ProtocolVersion { CASTV2_1_0 = 0; }
  required ProtocolVersion protocol_version = 1;

  required string source_id = 2;
  required string destination_id = 3;
  required string namespace = 4;

  enum PayloadType { STRING = 0; BINARY = 1; }
  required PayloadType payload_type = 5;
  optional string payload_utf8 = 6;
  optional bytes payload_binary = 7;
}
```

Message format

Source and destination fields used to multiplex messages
    Two pre-defined endpoints: sender-0 and receiver-0

Namespace field identifies message type.

Payload can be UTF-8 text (often JSON), or binary

Creating Channels

Before sending other messages, a connection must be established.

Namespace: urn:x-cast:com.google.cast.tp.connection
Payload: {“type”: “CONNECT”}

First thing sender does is establish connection from sender-0 to receiver-0

Many connections can be multiplexed over the message bus

Authentication

Chromecast uses a self signed TLS certificate, so includes a secondary authentication stage.

Sender transmits a challenge message, and Chromecast replies with response signed with platform key.

Only necessary if you want to ensure you're talking to a genuine Chromecast

- If you're not sending personal information to the device or doing DRM, can be ignored.

receiver-0 endpoint

urn:x-cast:com.google.cast.tp.heartbeat

Send: {"type": "PING"}
Receive: {"type": "PONG"}

urn:x-cast:com.google.cast.receiver

Launch and stop receiver applications
Track status of device (running app, volume level, etc)

Launching an App

Source: sender-0

Destination: receiver-0

Namespace: urn:x-cast:com.google.cast.receiver

Payload:

```
{
   "type": "LAUNCH",
   "requestId": 1,
   "appId": "CC1AD845"
 }
```

What happens on the Chromecast

Chromecast contacts Google server to retrieve app details:

```
{
   "resolution_height": 0,
   "uses_ipc": true,
   "background_mode_enabled": true,
   "display_name": "Default Media Receiver",
   "app_id": "CC1AD845",
   "url": "https://www.gstatic.com/eureka/player/player.html?skin=https://..."
 },
```

Launch returned URL in embedded Chrome browser

Launch Response


```
{
  "type": "RECEIVER_STATUS",
  "requestId": 1,
  "status": {
    "applications": [
      {
        "appId": "CC1AD845",
        "displayName": "Default Media Receiver",
        "isIdleScreen": false,
        "namespaces": [
          { "name": "urn:x-cast:com.google.cast.media" },
          { "name": "urn:x-cast:com.google.cast.inject" }
        ],
```

Launch Response (continued)


```
        "sessionId": "0BE51591-E561-4FD6-9D09-725C41C94B95",
        "statusText": "Ready To Cast",
        "transportId": "web-25"
      }
    ],
    "volume": {
      "controlType": "attenuation",
      "level": 1,
      "muted": false,
      "stepInterval": 0.05000000074505806
    }
  }
}
```

What happens on the Chromecast

Sender now has the endpoint ID to start exchanging messages with app

Knows message namespaces the app understands

What comes next depends on receiver application

Chromecast Apps

Can start any existing app.

https://clients3.google.com/cast/chromecast/device/baseconfig

Some well documented receivers:

“Default Media Receiver”

“Chrome Mirroring”

Create a custom receiver

Default Media Receiver

Implemented by Google

Many receiver apps are simply styled instances:

https://developers.google.com/cast/docs/styled_receiver

Google provides documentation for JS library for talking to media receiver, that tells us what it is capable of and how to use it:

https://developers.google.com/cast/docs/reference/chrome/chrome.cast.media

Chrome Mirroring

Special case: not handled as web content:

```
{
        "display_name": "Chrome Mirroring",
        "app_id": "0F5096E8",
        "native_app": true,
        "command_line": "/bin/logwrapper /chrome/v2mirroring --vmodule=*media/cast/*=1,*=0 ${POST_DATA}"
},
```

Seems to negotiate a WebRTC session using VP8 video and Opus audio

Some effort to reverse engineer:

https://github.com/thibauts/node-castv2-client/issues/14

Custom Receivers

Write HTML and JavaScript

https://developers.google.com/cast/docs/custom_receiver

Upload to HTTPS web site

Pay Google $5 to create a developer account and register application.

Writing an Ubuntu App

I wanted to be able to share videos and music from arbitrary apps to a Chromecast.

There are existing Cast protocol implementations for Node.JS, Go, etc

I wanted to write app with Qt/QML, so started work on a C++ implementation.

Writing an Ubuntu App (continued)

No mDNS responder on Ubuntu Phone

- Luckily Avahi provides an embeddable version as libavahi-core
- Implemented QAbstractListModel to browse services

Implement message bus in C++, exporting an interface to QML

Write the rest in QML / JavaScript

Still to do

Add an embedded web server to send content to the receiver app

Hook up to ContentHub

Try to get it working on the desktop too

- Talk to avahi-daemon rather than use avahi-core?

Other Project Ideas

Manage a collection of TVs at an event

- Show conference time table as HTML

Stream video or music to many locations at once

Use a custom receiver app to run code locally

Resources

C++/Qt:

https://github.com/jhenstridge/cast-app

Node.JS:

https://github.com/thibauts/node-castv2

Go:

https://github.com/ninjasphere/go-castv2

Xyz:

https://github.com/jloutsenhizer/CR-Cast/wiki