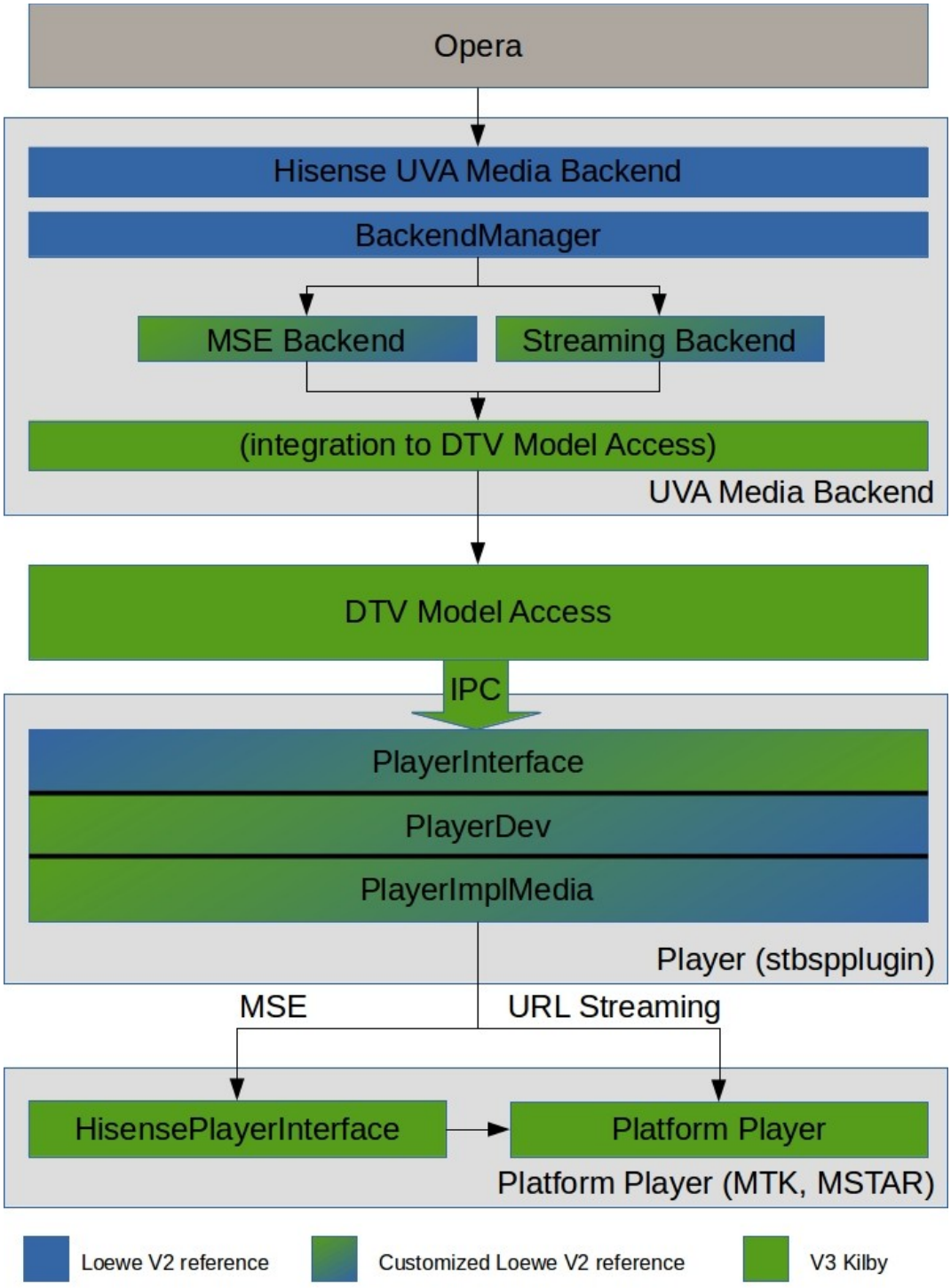
Backend Architecture and design

Table of Contents

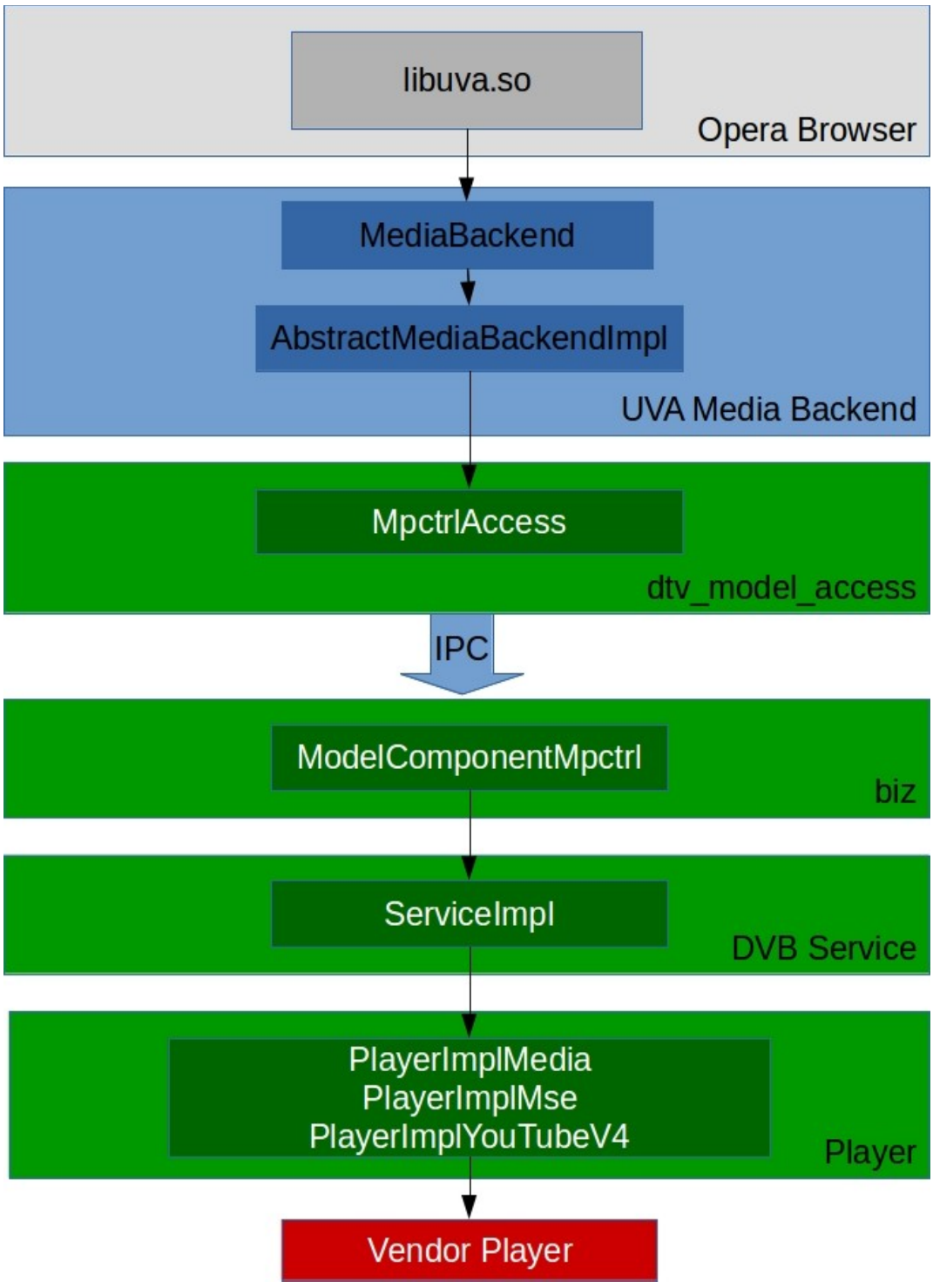
Phase 2 UVA Backend and Player Architecture

Phase 2 UVA Backend is based on Loewe V2 architecture modified to integrate with the new dtv_model_access. Player architecture is also based on Loewe V2 architecture (already modifed by Hisense) enhanced to add a new PlayerImpl implementation that integrates with a new Hisense player platform API (that all platform players implement).

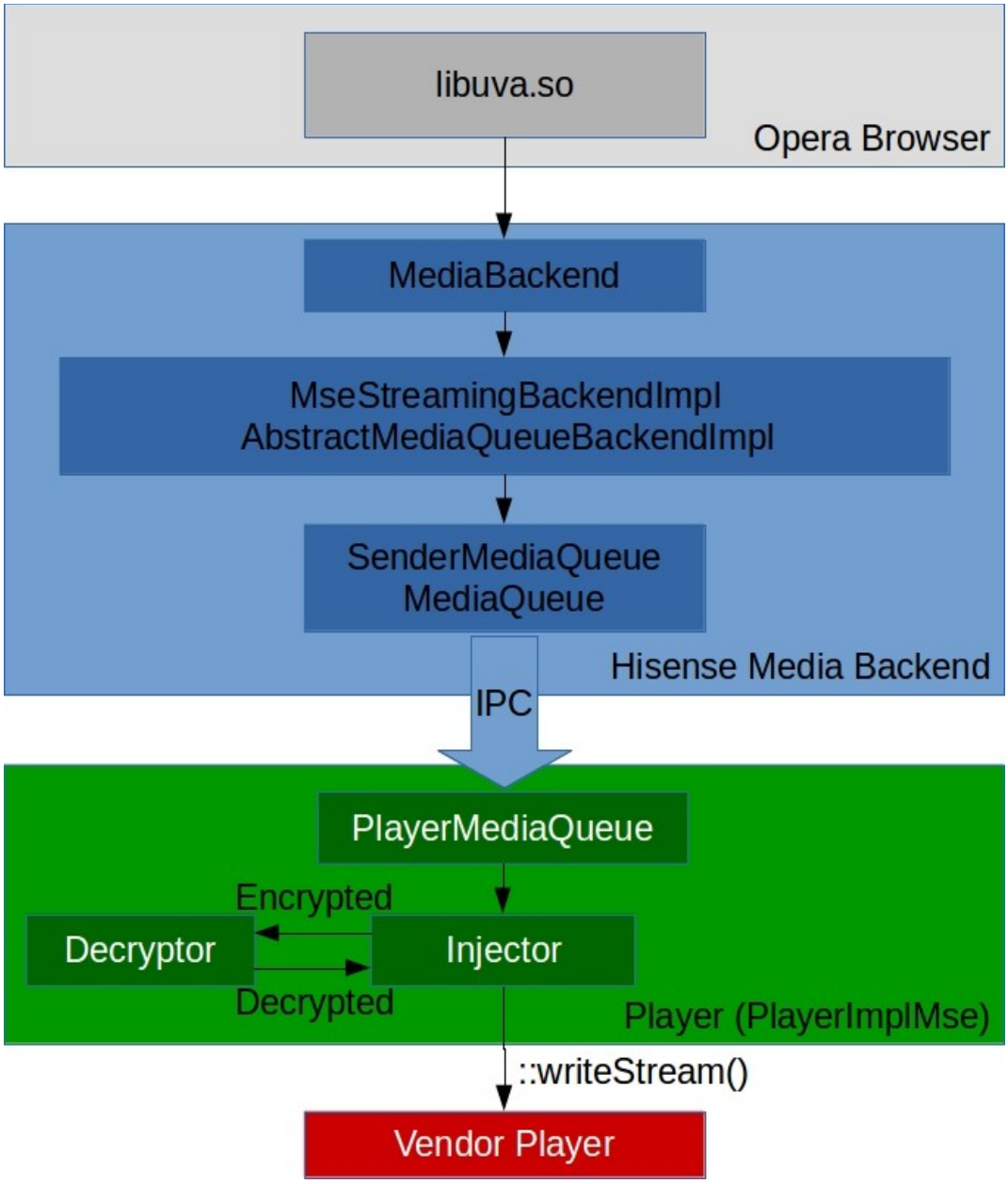
Component Interaction Diagram



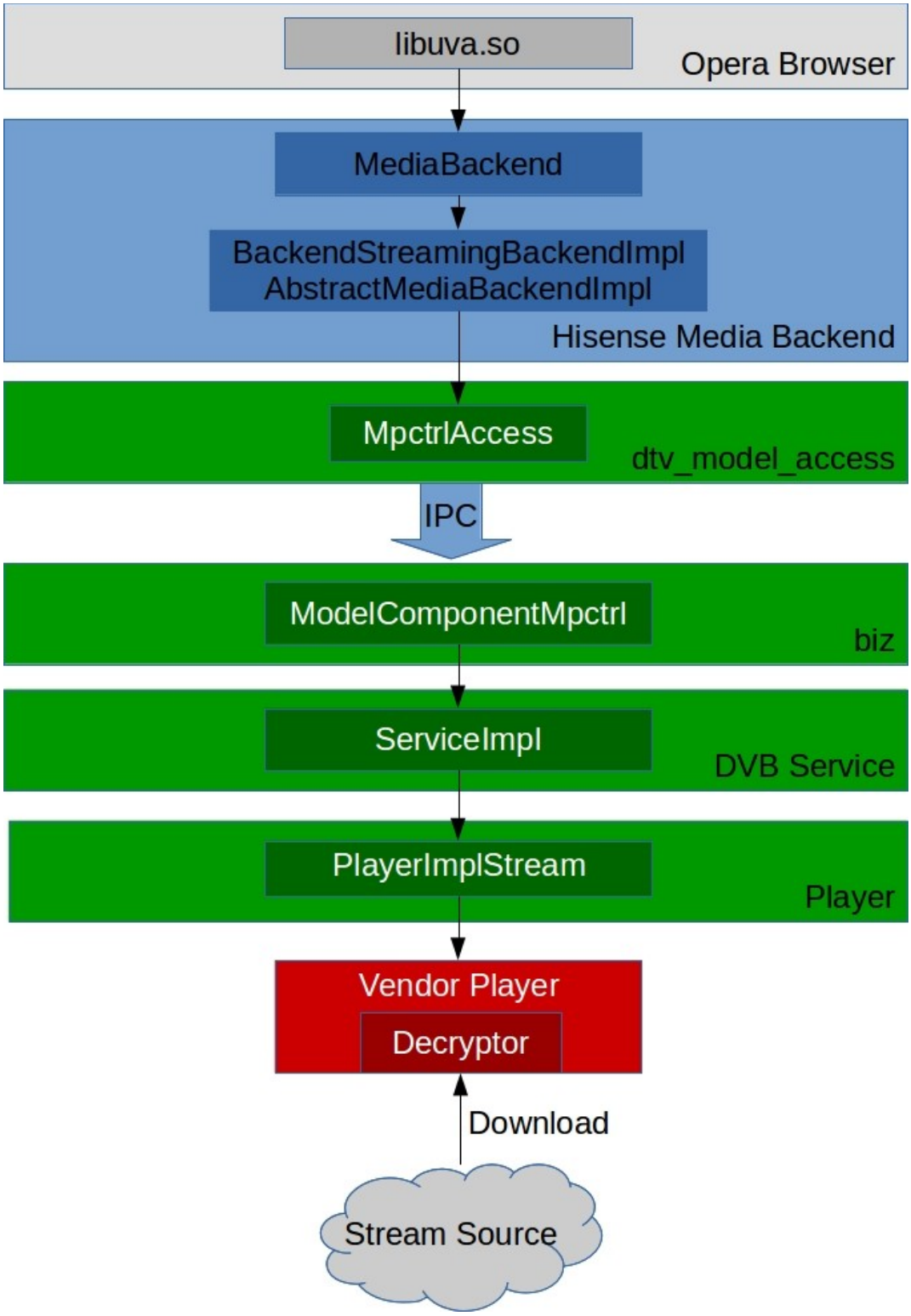
Command Path Diagram



MSE (and YouTube) Data Path Diagram



Streaming Data Path Diagram



Hisense UVA Media Backend Implementation

The new Hisene UVA Media Backend will be based on the Loewe V2 reference implementation. It will

be modified in order to access the model access implementations via the dtv_model_access component.

The current v2 loewe reference supports multiple instances of the UVA Backend with the BackendManager managing the multiple instances. Actions by an instance of a media backend that require the shared resources will result in all other media backend instances being suspended. When a media backend is suspended the media information (such as current position, volume, display rectangle, playback speed, and any other property required to resume playback as if it was never suspended) will still be maintained but all shared resources will be released. When a media backend instance is resumed while in a suspended state, all the previous payer info will be used to resume the player in the same state it was in prior to being suspended.

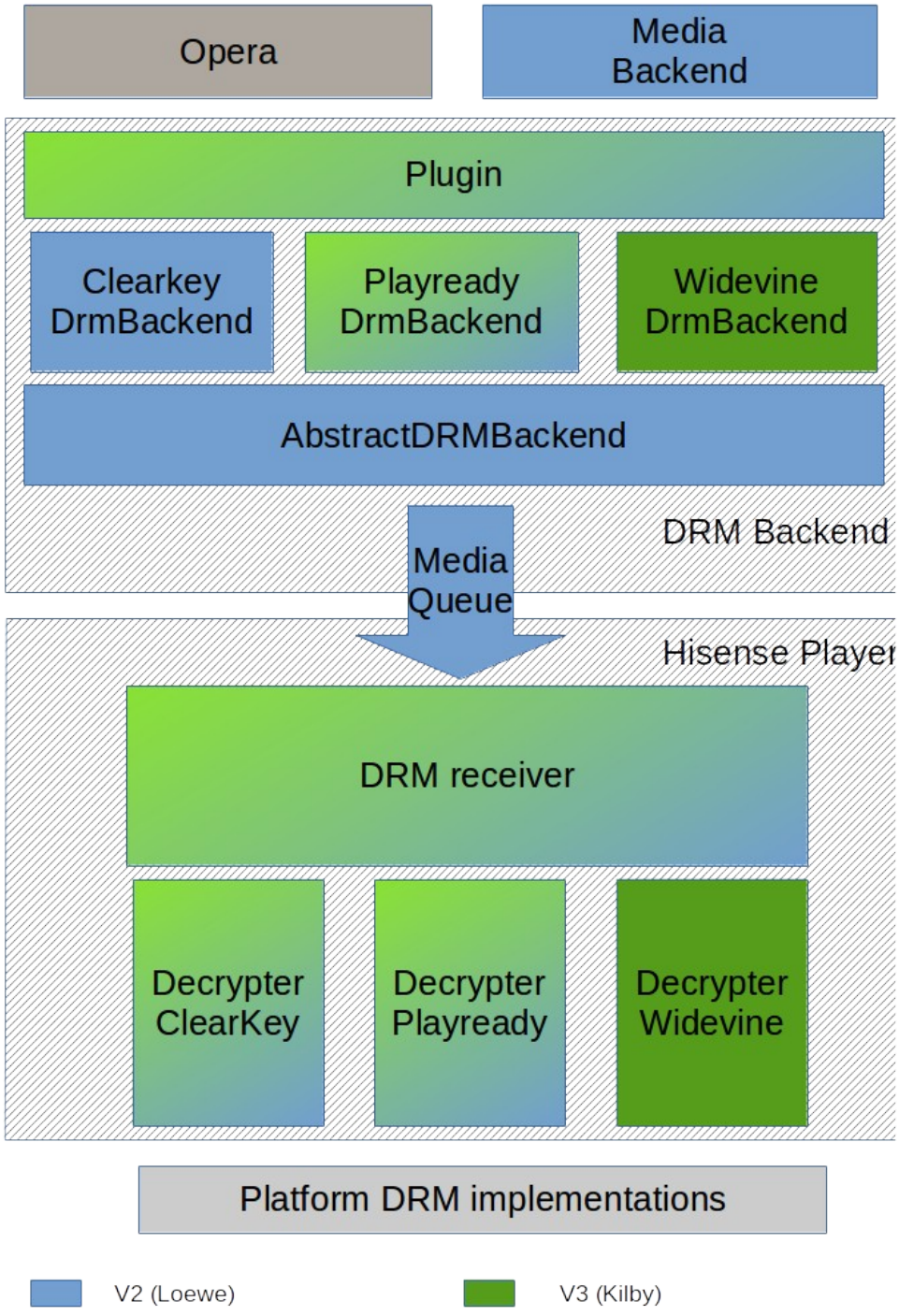
If supported by the platform player, a suspended media backend instance will still be able to send commands to the platform player while suspended to, for example, pre-load or begin buffering an MSE/streaming video.

Player Implementation

A modifed version of the Loewe V2 implementation of the player layer already exists in framework/coop/hilo/hl1/stbspplugin. The existing PlayerImplStream (for streaming) will be updated and a new PlayerImplMse (for MSE and YouTube streaming) will be created to integrate with the new platform player implementing the new player interface (described below). This new platform player API will be consistent across all platforms and chipsets.

Phase 2 DRM architecture

Phase 2 DRM architecture is based on Loewe V2 architecture modified for compatibility with Opera 4.7



For the Opera DRM backend the API part (plugin) is adopted for Opera 4.7 UVA DRM API

Playready backend may require some changes and Widevine backend is implemented using existing Clearkey and Playready backends as an example. Base AbstractDRMBackend and media queue implementation are left intact.

Player implementation

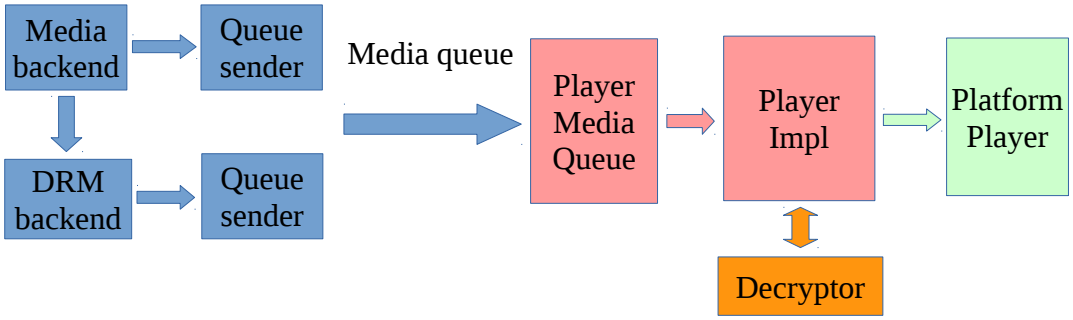
Player implementation is based on the existing Youtube player. DRM data and messages are received from media queue, then fed to one of the decryptors based on the DRM id. Clearkey and Playready decryptors can be inherited from v2 Loewe implementation, Widevine decryptor is implemented using existing decryptors as an example.

Current Youtube player contains some hardcoded code that depends on DRM type (such as setKey function), it must be moved to decryptors, the player should be completely neutral.

On the diagram above “DRM receiver” is the part of the player that receives data from the queue and directs it to one of the decryptors. Decryptors are using vendor-provided DRM implementation. It is expected that this implementation is providing the same API (DRM SDK) across all platforms, if this is not the case, some kind of HAL is required.

Data flow (v2)

Current (v2) data flow is based on sending media buffers from UVA backends to the player via mediaqueue. Buffered data is stored in userspace memory. There is a receiving thread which receives messages and pushes them to the player. The player injects media data into the platform player. When the player receives encrypted data, it is processed by a DRM decryptor, which returns unencrypted data in a userspace buffer.



For v3 / Phase 2 the requirement is to keep data decrypted by DRM in protected storage. This requires low level API implemented by both platform DRM and platform player for moving data in protected storage (TBD)

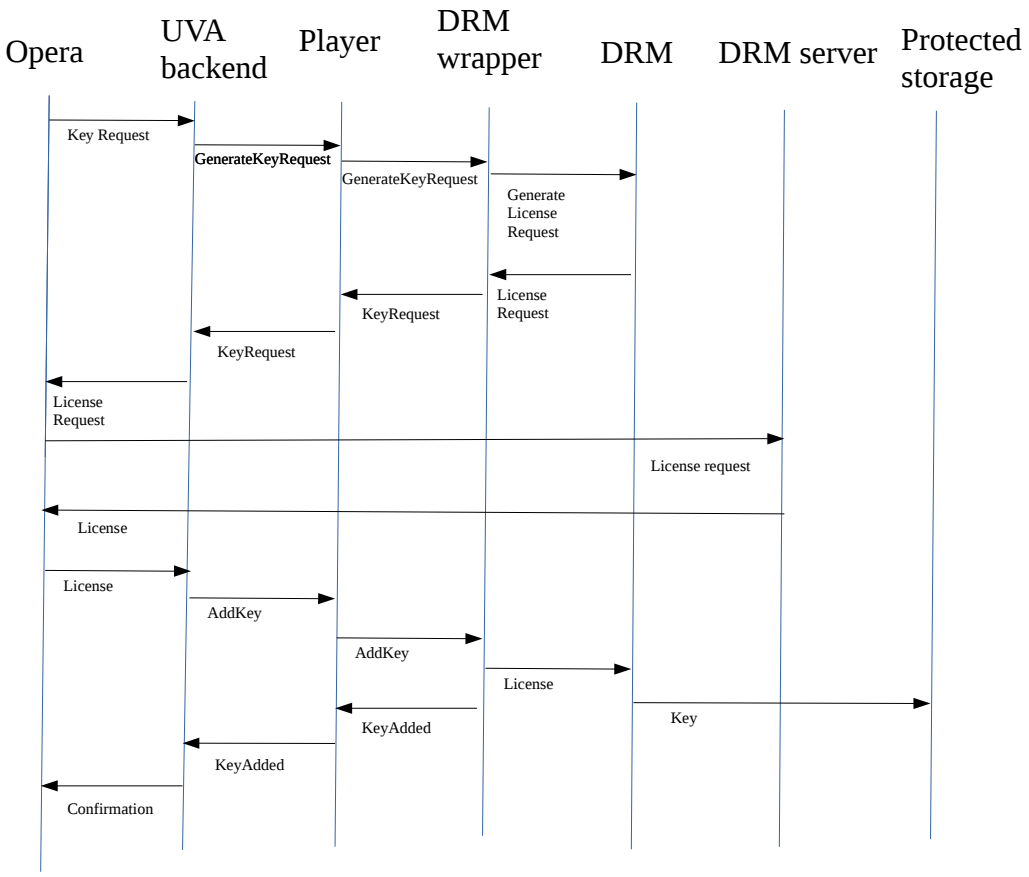
For unencrypted data, platform player accepts data in userspace buffer and dataflow architecture remains similar to v2.

Low-level API should use opaque buffer handles which can point to either userspace memory or in protected storage. Internal implementation of buffer handles is platform specific. Player implementation converts received messages to userspace buffers, which are either injected into the platform player as is, or sent to a DRM decryptor and converted to protected storage buffers.

DRM key flow

DRM key flow depends on the use case. Since different DRM systems provide different API and different usage model, there is a “DRM wrapper” class written for each of the supported DRM system that provides common API. This class is based on the existing “DRM context” model

DRM Key flow – EME



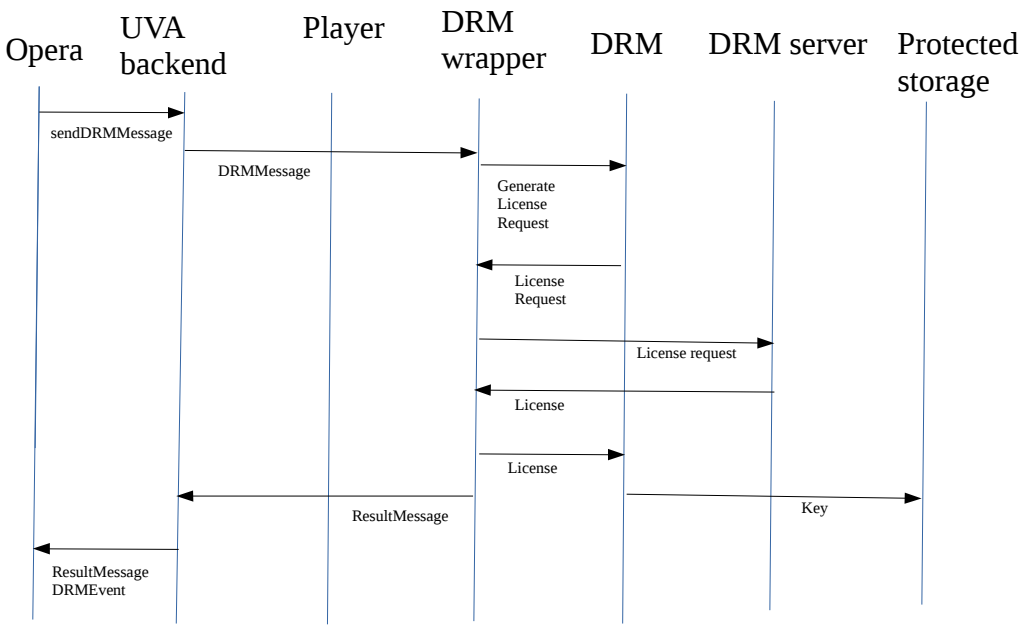
When playing media protected with EME, Opera is driving key acquisition. UVA backend is translating Opera request to a *GenerateKeyRequest* message, which is transferred to the player and makes DRM wrapper request DRM to generate license request. License request is returned back to Opera as *KeyRequest* and makes it request a license from the DRM server. The license is sent to the player in an *AddKey* message. DRM processes the license, decrypts the key and stores in the protected storage, then sends a confirmation message *KeyAdded* back to Opera.

In case of failure *KeyError* message is returned to Opera via UVA backend

DRM Key flow – non EME

When DRM is used for non-MSE playback and EME is not available, it is expected that playback is implemented by the vendor player and vendor player also takes care of DRM key management and decryption. This use case is out of scope.

DRM Key flow – DRMAgent



In the DRMAgent case (HbbTV) Opera is using *sendDRMMessage* method for sending messages to the DRM implementation. Those messages are specific for a particular DRM system and not interpreted by the backend. *DRMMessages* are transferred as is to the DRM wrapper, which takes care of parsing and executing them.

This diagram represents key acquisition messages, which makes DRM wrapper create license request (using DRM implementation) and send the request to the DRM server. The license is then processed by the DRM implementation and decrypted key is stored in the protected storage.

Note, that this is just an example, different DRM messages will result in different operations as supported by the DRM system.

After *DRMMessage* is executed, DRM wrapper generates response message (*ResultMessage*) which is transferred back to the DRM backend and returned to the browser as an *ResultMessageDRMEvent*.

Just like *DRMMessage*, *ResultMessage* format and content is specific to the DRM system.

Player Interface (Vendors)

The functionality required on the platform player is driven by the APIs that the new UVA Media Backend is required to support.

Required APIs

- play()
    - Starts (or resumes) playback at current position
- seek(int32_t position, bool pause)
    - Set playback position (seek)
    - position – position to seek to
    - pause – if true, seek to position but pause playback at that point
- getPosition()
    - returns current playback position
- stop(playerStopParams_t paramStop)
    - typedef struct _playerStopParams_struct {
      bool freeze; // if true, last frame is kept within the window, else a blank
      screen is shown
      } playerStopParams_t;
    - This parameter may or may not be required depending on front end requirements. Currently v2 loewe reference always seems to set this as FALSE
- pause()
- setSpeed(speedParam)
    - typedef struct _playerSpeedParams_struct {
      int32_t speed; // speed in percentage of the nominal speed (+100% is x1)
      } playerSpeedParams_t;
- setScreenPosition(lo::interfaces::playerRectangle_t pos)
    - typedef struct st_playerRect {
    - int32_t x;
    - int32_t y;
    - uint32_t width;

        - uint32_t height;
        - } playerRectangle_t;
- writeStream(stream_ptr stream, void *data, uint32 size, uint64 timeStamp)
    - write data (already decrypted) to stream
    - stream – stream to write data to
    - data – buffer handle, either userspace or protected
    - size – size of data in data buffer
    - timeStamp – set current timestamp for data
- dataRemaining(stream_ptr stream)
    - returns the size of remaining data in the stream buffer
- bufferSize(stream_ptr stream)
    - return the size of the buffer
- setTime(stream_ptr stream, time)
    - set current time position

The player will also need to support asynchronous events. Such events would include:

- current position changed
- duration has changed
- state changed
- AV component changed (e.g., audio, video, or subtitle track)
- AV component selected (e.g., audio, video, or subtitle track)