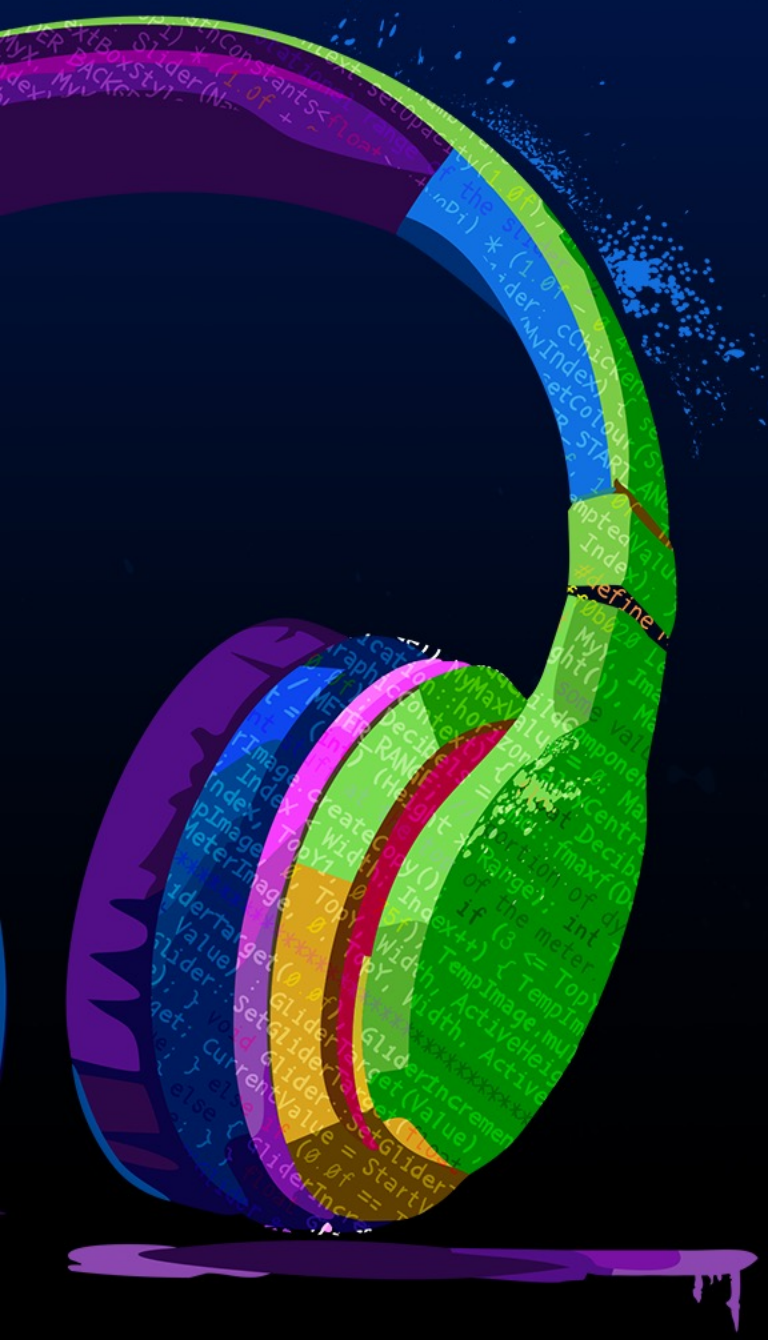


ADC24 Bristol

WHAT THE WORLD WOULD LOOK LIKE IF PLUGIN INSTANCES COULD SIMPLY TALK TO EACH OTHER

HOW WE APPROACH INTER-PLUGIN-INSTANCE-COMMUNICATION TODAY AND HOW IT COULD BE APPROACHED TOMORROW

JANOS BUTTGEREIT

Some sonible Slack internals

Agenda

sonible

- What is Inter-Plugin-Instance-Communication?
- How did we approach it until now?
- How could we do better with host support?
- Discussion

What is Inter-Plugin-Instance-Communication?

Inter-Plugin-Instance-Communication (IPC)

- Why do we even want plugin instances to communicate?
- Quick demo of sonible smart:eq 4

Lessons learnt from smart:eq 3

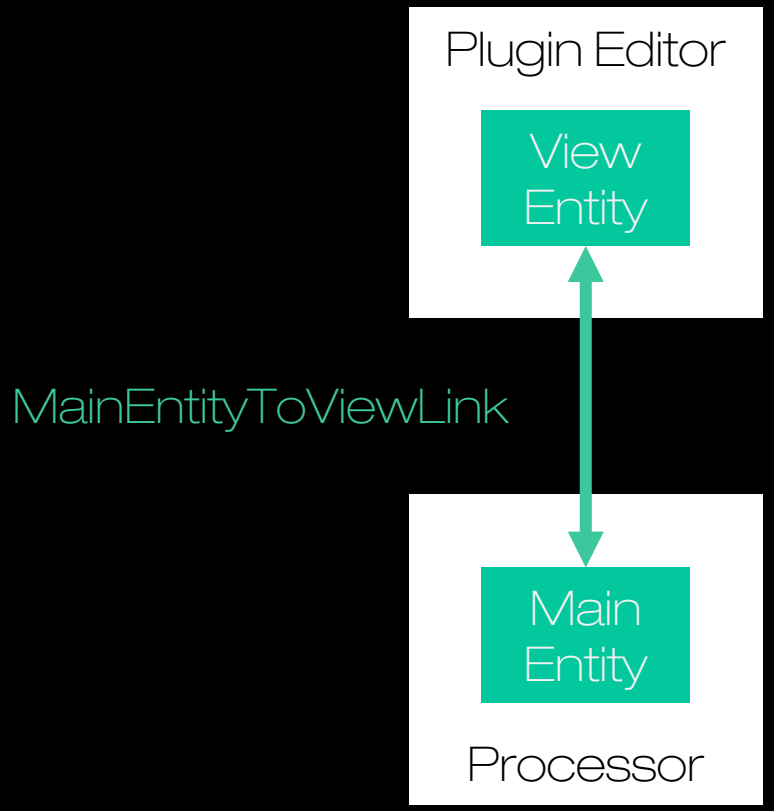
sonible

- You don’t simply add IPC at the end of the development cycle
- You should design your entire architecture with IPC in mind
- Abstraction, especially view/processor separation is key

Enter SPAPI (Sonible Plugin API)

- Abstracts a Plugin as a collection of connected Entities
- Example: Classic Plugin with one processor and one editor

Enter SPAPI (Sonible Plugin API)

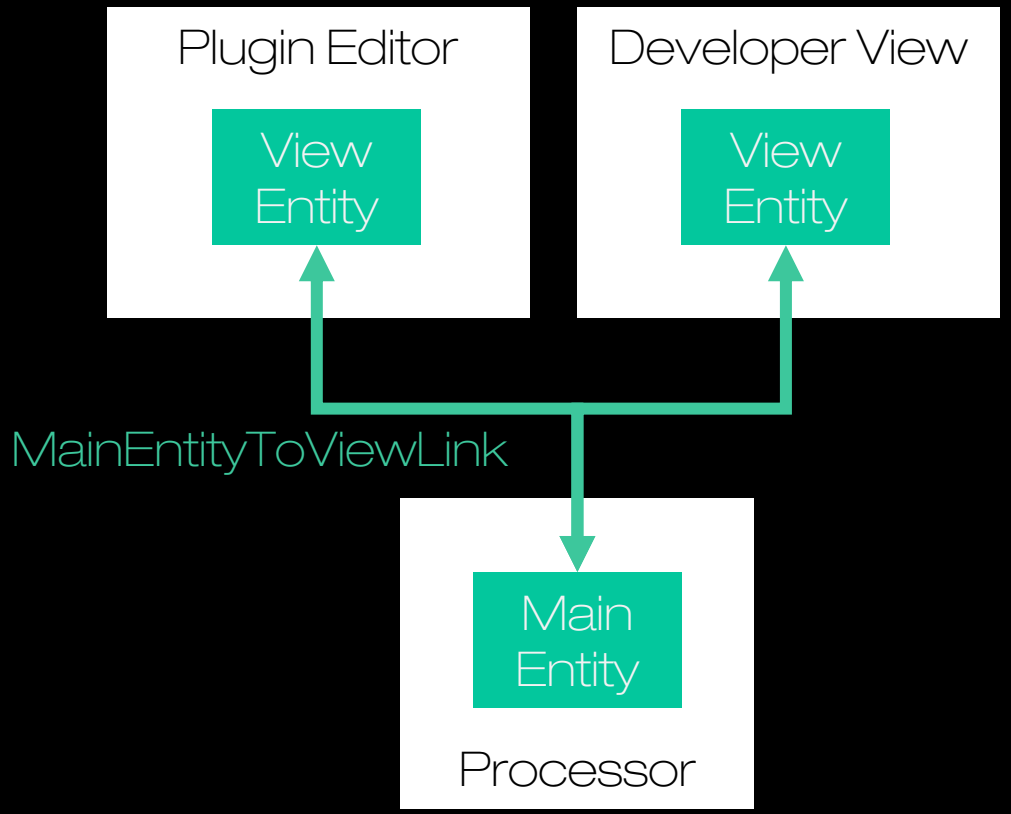


- Two types of *Entities*
  - *MainEntity*: Manages Parameters and Resources
  - *ViewEntity*: Read/Write access to *MainEntity* Parameters, read access to *MainEntity* Resources via *MainEntityToViewLink*
- A link can be used to send directional Actions and Streams

Core Features of SPAPI

- There can be multiple Views for a single *MainEntity*
- Example: Our generic developer view

Core Features of SPAPI

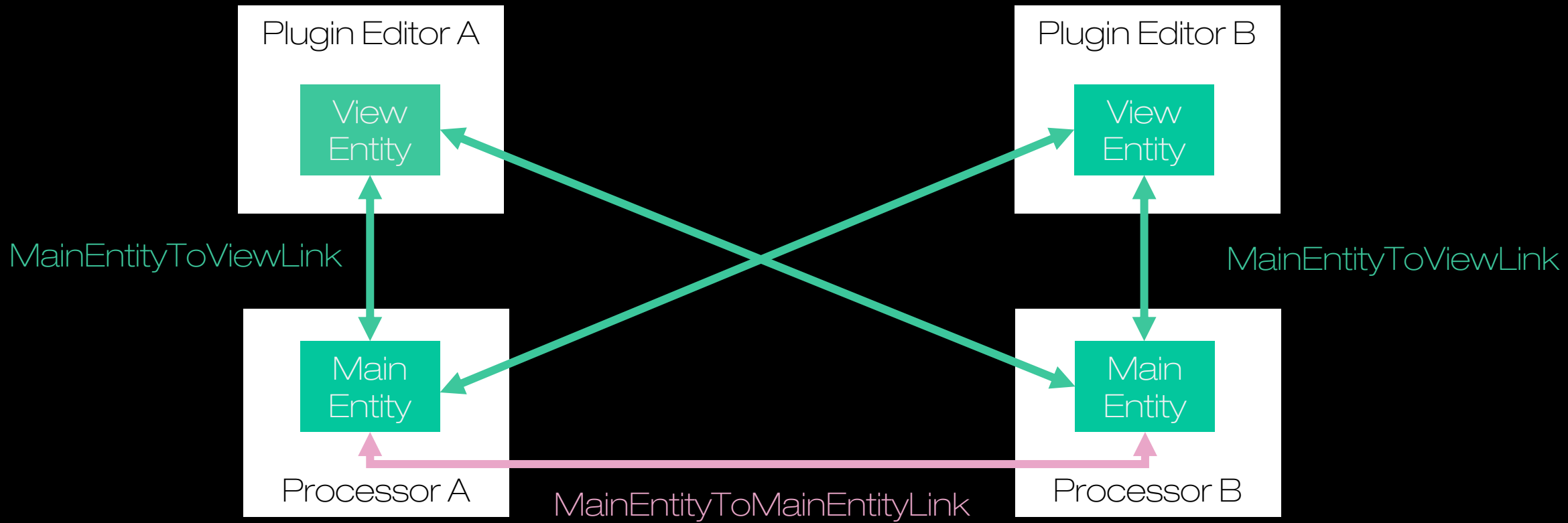


- All linked views are synchronized via the *MainEntity*
- Generic developer view is especially useful for development when the main UI is still under construction

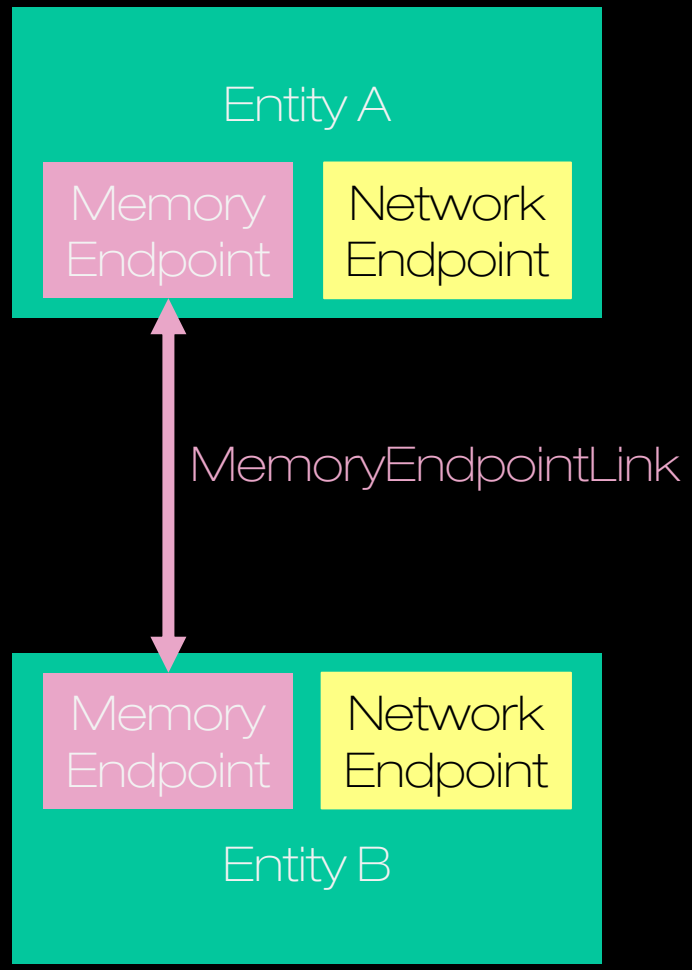
Core Features of SPAPI

- *MainEntities* can also be linked to each other
- Example: smart:eq 4

Core Features of SPAPI



Endpoint Types



- *Endpoints* implement different data transport approaches
- An *Entity* can contain multiple *Endpoint* types so that it can be reached from different contexts
- When establishing a link between *Entities* they check which *Endpoints* can be reached and pick the most suitable connection approach which defines the actual link type

Service Discovery

sonible

- How do Entities know about the existence of other *Entities*?
- The service discovery emits notifications when *Entities* appear or disappear
- At which scope should a service discovery operate?
- A stable service discovery is not trivial to implement

Group Management

- SPAPI has the concept of grouped *Entities*
- Only *MainEntities* can be part of a group
- All other *Entities* must be informed about the group
- Group management is tightly coupled to service discovery
- Reliable storing and restoring of the group state is quite challenging

How could we do better with host support?

Service Discovery

- We periodically send out UDP beacon messages from every *Entity*
- Service discovery instances receive them and update their internal list of known *Entities*
- If no beacon message was received from an *Entity* for a certain period of time, we consider it to have disappeared

```
message EntityDescription
{
    Identity identity                          = 1;
    HashedString wrapper_type                  = 2;
    HashedString bundle_id                     = 3;
    HashedString product_version               = 4;
    uint32 type                                = 5;
    uint32 ip_address                          = 6;
    int32 process_id                           = 7;
    HashedString executable_path               = 8;
    repeated EndpointDescription endpoints = 9;
    repeated NamedValue properties             = 10;
    repeated Group groups                      = 11;
    repeated Leadership leaderships            = 12;
}
```

Service Discovery

sonible

- It creates some runtime overhead to find out information that the host already has
- We apply filters to sort out *Entities* that are not hosted by the same process and/or are not hosted by the same application

Service Discovery

- It can fail in case
  - Multiple documents opened at the same time – Instances that don't belong to the project are found
  - Sandboxed plugins if we constrain it to filter entities by process ID – Instances that belong to the project are not found

Service Discovery

- We can only find out about the mere existence of an *Entity*
- We can’t find anything out about the role of the plugin in the processing graph – the user has to make sense of that manually

Service Discovery

sonible

What if there was an API to query information about other plugins and at best, the structure of the entire processing graph of the current document from within the plugin?

Endpoints and Links

sonible

- We filter by process ID → In-process memory based *Endpoints* are used
- Implementation relies on a static map
- Generally works but fails for communication between *Entities* from plugins of different types – even wrapper types

Endpoints and Links

What if there was a host supplied data channel to send data from a plugin instance to another?

Group Management

- Session persistent links and *Entity* groups are a session wide state
- Host only allow to save a per-plugin instance state

Group Management

sonible

- Be prepared for nasty edge cases
  - What if a instance plugin has restored its group state before the group members have been re-created in the session?
  - What if a plugin instance is copied by the user and with that the state is duplicated?
  - What if there are DAW maintained plugin presets?

Host managed Plugin Presets

- Support Ticket: Group management goes crazy when the user applies a plugin Preset in Reaper

Host managed Plugin Presets

sonible

I fixed it.

```
void restoreEntityState (const proto::state::PluginState& pluginState)
{
    // Super hacky workaround to prevent loading an entity state when reaper
    // applies a preset to the plugin
    if (juce::PluginHostType().isReaper() &&
        juce::SystemStats::getStackBacktrace().contains ("fxLoadReaperPreset"))
        return;


    // continue restoring the state
}
```

Group Management

sonible

What if hosts would either have built-in group management or would allow us to install our own centralized group management facility in the project?

Interim Conclusion

- We would like to have (yet another) extension API that allows us to implement IPC functionality without hacky detours
- At best this API would be as independent as possible of the plugin wrapper format – but also format specific extensions would help us

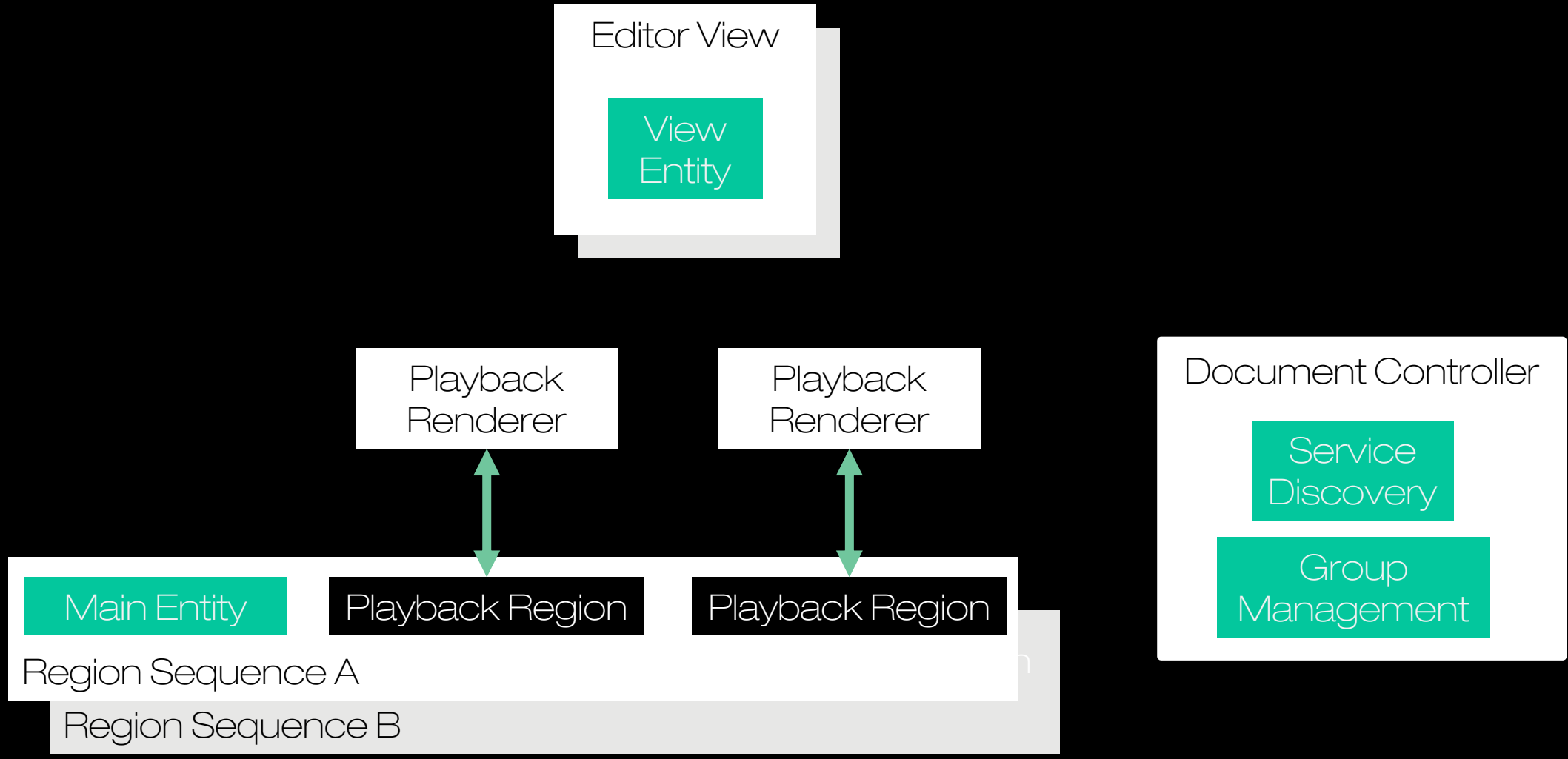
How could an IPC API work and look like?

A look at ARA

sonible

- ARA is a good example of a successful companion API
- Partly implements the desired pattern
  - Abstraction of the document
  - Single management instance (Document Controller)
  - Separate renderer and view instances

Some ARA Model objects



So ARA already solves the problem?

- ARA solves the problem only for a specialized use case
- No solution for generic plugin use cases
- No abstraction of the rendering graph, document abstraction stops after playback renderer

→Still a good example of how an extension API can be approached

How could an IPC API work and look like?

- Multiple *Entity* classes are needed
  - *ViewEntities*: Plugin editors etc. → Everything that is intended for user interaction
  - *MainEntities*: Plugin processors, document controller – everything that actively manages a stateful functionality of the document and is available for connection
  - *PassiveEntities*: Everything that contributes to the processing graph etc. and can be queried but is not available for connection

The Service Discovery

Listeners that are notified for specific *Entities* can be registered.

```
// This handle will be injected via e.g. a plugin format extension
auto* ipc = getIPCHandle();

if (ipc != nullptr)
{
    auto& disco = ipc->getServiceDiscovery();


    // Get notified about all Main Entities of a certain plugin (dis)apppearing
    disco.addListener (this, ServiceDiscovery::Filter()
                             .withEntityKind (Entity::Kind::mainEntity)
                             .withBundleID ("com.sonible.smarteq4"));
}
```

Linking Endpoints – Option A

Links could be a transparent channels for binary data blobs. The protocol can then be manufacturer specific

```
auto& myEntity = ipc->getMyProcessorEntity();

// Called whenever an entity sends data to this one
myEntity.setDataHandler ([] (std::span<const std::byte> data,
                             const Entity& source) { /* handle */ });

// otherMainEntity is an entity object retrieved via the Service Discovery
auto link = myEntity.connectToAs<LinkToMainEntity> (otherMainEntity);

if (link != nullptr)
    link->sendData (...)
```

Linking Endpoints – Option B

Links could also be used to directly access a predefined set of data structures and interfaces

```
auto& myEntity = ipc->getMyProcessorEntity();

// otherMainEntity is an entity object retrieved via the Service Discovery
auto link = myEntity.connectToAs<LinkToMainEntity> (otherMainEntity);

if (link != nullptr)
{
    // Get notified if the linked entity changes its resource "preset"
    link->resources().addListener (resources::preset, [] (auto& p) { /*...*/ });
     // Trigger the action "start_learning" on the linked Entity
    link->actions().send (actions::start_learning);
}
```

Group Management Options

- Multiple different approaches could work
  A. Plugin can install an ARA document controller like group Management instance → Extra entry point in plugin binary
  B. There is a new kind of plugin: IPC group management plugin
  C. IPC API has built-in group support. The host manages the groups, stores and restores them

Group Management Options

- How sophisticated should groups be?
- Where should e.g. inter-group computations happen?
- Should different plugin types be grouped together (e.g. EQ and compressor)?

How to settle on a standard?

- So we have a lot of requests and questions towards host developers
  - Where and how to discuss them best?
  - How to settle on a standard?

Let's discuss

Or contact me later:

janos.buttgereit@sonible.com