Weston surface remoting for IVI

Collabora, ADIT

Pekka Paalanen

Table of Contents

1. Introduction

In the foreseeable future some embedded systems, In-Vehicle Infotainment (IVI) in particular, may start running multiple operating system (OS) instances for different “domains”. These instances may run on a dedicated hardware unit each or virtualized on a single hardware unit. Sometimes programs in one OS instance may need to present windows in a different instance with both (user) input and output (display).

For IVI these domains are usually the automotive domain which drives the instrument cluster and other safety critical features, and the multimedia domain which drives the center console, plays music and video, runs the navigator, and so on. Reasons for sending a window across the domain and OS boundary can include showing a navigator view in the instrument cluster or on auxiliary entertainment displays.

This proposal is about designing a protocol for redirecting both surface (output) and input from one Weston instance to another over a) network, b) virtualized memory-sharing transport. Output remoting will happen on pixel buffers with completed rendering, not by sending rendering commands to the remote. The design is aimed for IVI, but leaves room for augmenting it to support desktop shell protocols if wanted.

To minimize impact on applications, remoting will be implemented in the Wayland compositors (Weston) transparently to any applications. Presumably there is no need to *require* any Wayland protocol extensions to be used by the clients, but there might be use for e.g. content type hints, in case there will be several kinds of remoting (e.g. different video codecs or profiles).

Wayland is not a network-capable protocol, therefore Wayland will be used only as an inspiration and a model, not an actual transport.

Remoting will be started by the local compositor connecting to a remote compositor, and pushing a local client surface to the remote. The opposite is the pull model, where the remote compositor connects to the local compositor and asks for a specific surface to be sent to the remote. Pull model will not be supported at this level, but it can be implemented with the IVI controllers, where the remote controller sends a request to the local controller to push a surface to the remote.

As resources for remoting are limited, it is assumed that the use cases have been designed to avoid exceeding them even in the worst case scenario. These limits include the number of remoted surfaces which is tied to the number of simultaneous video encoding and decoding contexts, and the total network bandwidth required. Exceeding the limits may cause remoting to fail to start, remoting to stop unexpectedly, or reduced framerates and responsiveness on remote surfaces.

Further assumptions on this work include stable and practically guaranteed network conditions, latency and bandwidth. This is generally not true over the Internet, but should still offer a viable experiment on how a protocol similar to Wayland would fare over network.

1.1. Out of scope

The following items are not considered in the design. Most of the feature additions could be implemented or added to the design without major architectural rework, but there are no guarantees

on how well they might fit.

- clipboard support
- drag & drop support
- input methods
- input devices other than the Wayland core wl_keyboard, wl_pointer, and wl_touch
- any inter-client communications
- shell protocols other than implemented in Weston's ivi-shell
- protecting compositors from network denial-of-service or video encoding/decoding issues
- audio and AV-sync, multimedia playback
- remote rendering (submitting drawing commands instead of completed images)
- not using Weston as an end-point
- surface remoting started by pull-from-remote (the pull model)
- showing a surface on more than one compositor
- communication between IVI controllers in general, except for identifying the pushed surface
- overcoming hardware limitations on how many surfaces can be incoming and outgoing at a time (which can be a very low number like 1 or 2 for some hardware)
- configuring network QoS or testing it (depends on network topology and number of ECUs)
- dynamic rate control of video streams
- multicast (same surface to several remotes taking advantage of multicast)
- having different encoding profiles for different content types

Support for sub-surfaces or Presentation extension will probably be left for later.

2. Architecture

2.1. Overview

Each domain/OS (or virtual machine) called “ECU” has a Weston instance running. Weston has a plugin and a helper process for remoting which include everything network and video compression related.

An overview of the architecture is given in Figure 1. The IVI application is running in ECU A on the application-side or source compositor. A surface from the application is remoted to ECU B, the destination compositor which is also called the display-side compositor for the remoted surface. The application receives input events from ECU B's input devices as appropriate. The compositor roles are named with respect to the remoted surface, but all compositors can act both roles even at the same time when multiple surfaces are remoted in both directions.

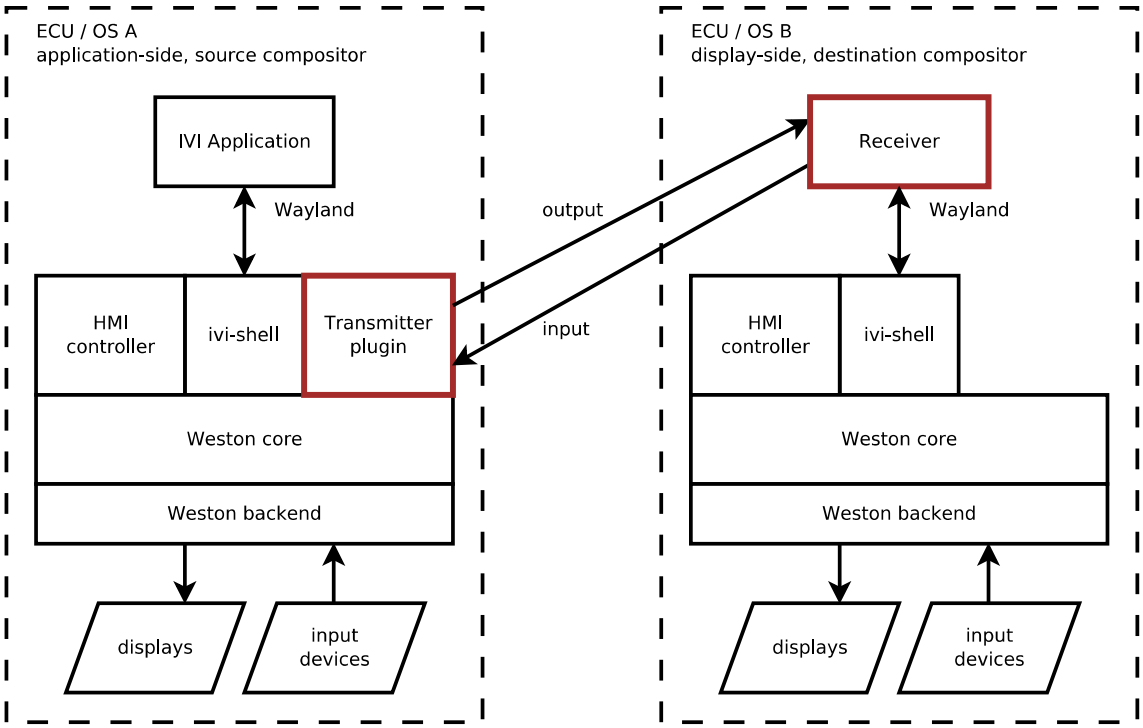

Figure 1. Architecture overview from the point of view of a single IVI application showing on a remote compositor. The thick red components are to be developed new.

The compositor setup is asymmetric. The source compositor needs Transmitter plugin for catching the surface to be remoted in ECU A. Transmitter connects via network (or other means) to Receiver in ECU B. Receiver is a helper process: a client to the destination compositor, and a network server listening for incoming connections from other compositors. Receiver uses the usual Wayland protocols to replicate the remoted surface on the destination compositor and get input events to be passed back to the source compositor. The source compositor does not require a helper process unless it acts also as a destination compositor, and the destination compositor does not require Transmitter plugin unless it acts also as a source compositor. It is possible that launching Receiver from the destination compositor could use a trivial plugin. The asymmetric design allows a compositor to be configured either as a source or a destination, or both at the same time.

Transmitter has to be a Weston plugin, otherwise it does not have access to the compositor internals required to “hijack” surfaces to be pushed to the destination compositor and provide input events for the surfaces. Transmitter plugin will create `wl_output` and `wl_seat` Wayland protocol objects to represent the remote input and output, but it cannot be a Weston backend, because a normal backend is needed to manage the local input and output devices. Transmitter plugin may also need to be able to create `weston_view`s for remoted surfaces without putting them into the local scenegraph and gain access to the `weston_surface::configure` hook, but it cannot be a shell plugin, because a normal shell plugin is required for managing the local surfaces and views.

Receiver is not a plugin but a separate process and a Wayland client to the destination compositor. Receiver needs to create buffers and fill them which are best done as a client. If it was a plugin we would have to add support for internal buffer allocation which would probably touch much of core Weston. Downsides of being a separate process are likely limited to just CPU context switches and the Wayland protocol overhead which should be negligible compared to the cost of going over network. For the virtualized case, profiling will be needed to determine if having receiver as a

plugin could yield any performance improvement. The main benefit of a separate process is that no modifications to Weston are needed for Receiver, the usual Wayland protocols should be enough for showing remote applications. Other upsides are that the network server part will be isolated from the compositor, which improves stability, and Receiver could run on other compositors than Weston.

2.2. Window management

Window placement must be controlled by the compositor displaying the window. With remoting, this is the destination compositor. Only the destination compositor has a full view of what is on its outputs, and can decide what and where to show (ignoring a centralized Layer Manager here). This is especially important when the destination compositor is in the Automotive Domain (AD), as it has safety critical displays that no other sub-system must be able to mess with.

To be able to manage a remote surface, the destination compositor needs the window metadata from the source compositor's shell. The metadata is shell specific, and must be transmitted. This calls for a **window metadata protocol** which would be a part of the remote control protocol. For IVI, the window metadata protocol carries the IVI-surface ID and possibly a resize event (`ivi_surface.configure` from `ivi_application` protocol). The window metadata protocol must be replaceable as e.g. desktop shell would use different messages. Both end-points are assumed to speak the same protocol, otherwise remoting will fail.

2.3. Controlling remoting

Remoting a surface is initiated by the application-side (source) compositor. It connects to a remote compositor and pushes surfaces with window metadata there. The destination compositor may accept or reject the push.

In the source compositor, the shell makes the decision (or receives orders) to remote a surface. In IVI-shell's case IVI-layout API shall have functions to control remoting of an IVI-surface, which IVI-layout translates to Weston Transmitter plugin API. This puts the IVI controller module in charge, which is assumed to know when and what to remote. Similarly in the destination compositor, the IVI controller is assumed to know how to deal with incoming remote surfaces based on their IVI-surface ID.

The push model is chosen over a pull model or a mixed model for simplicity. A generic pull approach would require updating window lists for the remote so that the remote could pick some to pull, though in IVI environment the IVI-surface IDs would be known by design without dynamic discovery. It seems the push model would be more generally useful, and more close to the traditional Wayland client model. The receiving compositor (display-side) can then decide whether to show it or not, or even reject that surface.

2.4. Networking vs. virtualization

Virtualization is not considered at first as the platform is undecided. There is too much unknown in what the platform will offer by means of virtual transport, e.g. what kind of shared memory is available and can it be used directly by hardware units like the GPU.

Any thoughts towards virtualization in this design assume that there exists a fast virtualized transport offering zero-copy operation, which allows passing raw pixel buffers from one OS to the other. It is important for performance, that the GPU on both sides can directly render to and texture from this virtual shared memory. As there is no need for encoding or compression, GStreamer will not be needed in this case.

The control channel described for networking in Section 4 is also applicable for the virtualized case. Using network for the control channel should have a negligible overhead compared to any other “more virtualized” approach. Therefore all plans for input events, surface meta-data, etc. control communications are directly usable for virtualization.

3. Detailed notes

This section contains some random thoughts and rationale on the intended implementation.

3.1. Weston implementation

We will have the general problem of Weston plugins needing to call API offered by other Weston plugins (ivi-shell vs. remoting plugin). [get vfunc table based on string name and major version, minor version is vfunc table size]

Transmitter plugin has backend features: it will create outputs and seats with input devices, and it needs it's surfaces away from the scenegraph but still requires the renderer in order to gain access to surface contents. The input devices need to bypass Weston core and provide input directly to the designated target surfaces, as the input device foci are controlled by the remote. The outputs cannot be part of the global coordinate space nor managed by Weston core, because that would imply the source compositor being in control of positioning and the output.

Weston needs a way to force an output for a surface to sync to. The usual output assignment code cannot handle outputs that are not part of the global output space.

3.2. Output remoting

All video encoding is driven from the compositor process, and so is network communications for Transmitter. Video decoding happens in the Receiver process.

Should remote outputs be represented in a local compositor as `wl_output` objects towards the local clients?
- Probably yes, so that `wl_surface.enter/leave` can work and convey output scale and orientation.

Presentation extension? Requires clock synchronization, which can be achieved with external NTP or PTP software, and converting between the synchronized clock and a compositor's graphics clock. Presentation time is determined as usual: by the output the surface is synced to. When remote outputs are involved, the shell needs to set which output to sync to.

Probably want to be streaming frames; latency can be higher than frame period, but still want to present at native framerate. How to trigger `wl_surface.frame` callbacks? An asynchronous clock based tracking of remote framerate and timer in the local compositor to mirror the remote refresh cycle with after-the-fact timer synchronization? Or just a simple throttling scheme in case the network bandwidth runs out, based on acks from the remote and buffer send queue length?

If the video encoder or network link cannot keep up with the window updates, frames may be dropped. Frame dropping both before encoding and in the remote compositor will eventually cause the Wayland frame callbacks to be throttled accordingly, and this will automatically throttle the application. How?

Even when frames are dropped, the local compositor will ensure that the last application committed frame will eventually reach the remote compositor, unless replaced by another frame, the surface is destroyed, or remoting is stopped.

A surface cannot be visible on both local and remote displays at the same time.

Does Transmitter need the `wl_buffer`? At first no, it will ask Weston's renderer for the surface contents, so there are no side-effects for buffer management. The renderer might also give out a buffer reference when it is not maintaining a copy of the surface contents. To avoid `glReadPixels()` when optimizing the buffer handling, Transmitter may need to keep its own reference on `wl_shm`-based buffers. Likely the renderer will abstract buffer types into something that makes sense for GStreamer.

3.3. Input remoting

Input comes directly to the compositor from the network.

What level of protocol to relay?
- evdev, libinput? No, the remote compositor needs to filter the event stream, like for any client.
- Most likely the remoted input protocol should look a lot like the Wayland input protocol. The sending compositor decides which surface they target, computes surface coordinates, etc.

Should remote input devices be represented in a local compositor as `wl_seat` objects towards the local clients?
- Yes. Representing remote seats as new wl_seat objects is necessary to avoid confusing input events and foci. It will be easier to manage.
- Will need a Weston core bypass path to be able to feed input events directly from network to clients.

Keyboard events will be sent identical to Wayland: key codes as defined in Wayland, plus keymap data.

4. Network transport

Network transport consists of two main pieces: Wayland-like control messaging, and bulk pixel (video frame) transfer which uses compression/encoding.

4.1. Channels

Network connections are point-to-point and independent, a Transmitter (network client) connecting to a Receiver (network server). Remoting a surface uses several high-level *channels* (TCP or UDP). There are two types of channels in general:

1. **control channel**: Low latency, high priority, short messages; for surface meta-data, window management, and input.

2. **data channels**: Fairly scheduled among each other, bulk data transfers; these are created primarily for video data streaming.

The control channel transport must guarantee lossless, ordered message delivery. The API must be asynchronous. Sending a message must not imply a roundtrip or block the caller. The protocol in the control channel is almost the same as Wayland except file descriptor passing is not supported. C API for protocol is generated from XML files similar to Wayland. The messages on the control channel are short and do not contain pixel data. Medium-sized payloads like keymaps can be delivered on the control channel at first. If the need arises, medium-sized payloads could be off-loaded to new data channels. It is expected that the most urgent messages will be the input events and the output feedback sent back to Transmitter.

There are several possibilities how a control channel could be implemented. The two main choices so far are extracting the protocol and network layer from an earlier project at Collabora, and forking libwayland and modifying it to use TCP and remove unnecessary features. On top of that, a C bindings generator similar to `wayland-scanner` is needed.

The data channels are mainly used by GStreamer elements as part of a remoting pipeline. The actual TCP or UDP connections are a detail of the GStreamer framework and there can be several for a single remoted surface. These need to be taken into account with the network QoS configuration.

Network Quality-of-Service (QoS) is used for scheduling the channels. Control channel could be prioritized over data channels up to a percentage of network bandwidth for example. All data channels together could be allocated a percentage of network bandwidth while being serviced in a round-robin fashion. This can be realized by type-of-service flags in IP or specific QoS rules. See http://www.lartc.org/howto/.

Note, that we consider only a case of two ECUs with essentially a direct link. If the network topology is more complex, then the QoS rules and particularly network bandwidth allocations must be revised. QoS configuration is not considered here apart from the above hand-waving.

4.2. Input event transport

Input events will be passed in the control channel as is. The protocol will mirror the Wayland input protocols.

Input events are simple to relay, there is no out-of-band payload involved. Also input event timestamps should be fine without any clock conversions, because Wayland does not define what base the input clock has. Still, changing clock base may cause glitches at the time an already visible surface becomes or stops being remoted. The glitch is avoided if a surface is visible only on one compositor during its lifetime.

4.3. Pixel transport (output)

4.3.1. GStreamer, RTP, and Farstream

GStreamer will be used for the surface content transmission as a video stream. GstRtpBin offers a ready solution using RTP and RTCP, and GStreamer elements for hardware accelerated encoding

and decoding can be used.

GstRtpBin can be used through Farstream. Farstream is a framework offering many features for internet conferencing including NAT-traversal, but the most useful feature in this project is automatic CODEC negotiation.

GstRtpBin comes with many useful and optional features that would be hard to obtain otherwise, including retransmission, encryption, and synchronization. Synchronization can also be disabled for displaying as fast as possible. It does not require a fixed framerate, RTP being purely timestamp and sequence based.

A downside is that it is not possible to handle only damaged rectangles. Choosing the right compression method or CODEC will mitigate this.

4.3.2. Encoding and image quality

Surface content will be encoded into video frames only when the content changes, that is, a Wayland application commits a buffer to a `wl_surface`. If there are no changes or commits, no frame will be received by Transmitter, encoded or transmitted. This requires the encoder to have a zero-latency setup. The benefit is very small network bandwidth usage when the image is static. Encoding and transmitting will follow the Wayland application frame rate up to a maximum possible throughput.

While remoting is transparent to Wayland applications in theory, it may cause the compositor to hold on to the `wl_buffers` longer than usual. A video encoder may require several old buffers to encode a new one. If also a zero-copy path from an application to the encoder exists and is used, then the compositor must hold on to the buffers as long as the encoder needs them. This is visible to the application in that `wl_buffers` do not get released as soon as usually expected. If the application, toolkit or e.g. libEGL does not handle this case by allocating more buffers, the application may freeze or crash. However, this is considered an application bug, as Wayland makes no promises on releasing buffers.

Image quality will mostly depend on encoder settings, but requiring zero-latency encoding for timely frame transmission probably reduces the quality somewhat. Image quality issues may be exacerbated by the on-demand nature of encoding: regardless of the extents of changes in the image, they will be encoded over a single frame only.

If image quality is not sufficient after simple tuning of encoder settings, it is possible to switch the design to use a constant framerate encoding. If the image is not changing, the same image gets fed into the encoder and transmitted repeatedly. This leads to increasing image quality in the destination over the time the image stays static, but consumes some network bandwidth even when nothing is happening with the application surface. Constant framerate encoding could also be used to work around a missing zero-latency mode of an encoder, if necessary.

Remoted surfaces may have different types of content. User interface graphics are likely more susceptible to compression artifacts being visible than, say, rear view camera feed. It would make sense to have different content types encoded with different settings. Tuning encoder settings, adding alternative encoding profiles, and letting applications tag different content types is left for

future work.

4.3.3. Multi-instance CODEC

Each remoted surface occupies one encoder context in Transmitter and one decoder context in Receiver. Using a separate data channel for each surface lets them update without starving each other on the network, assuming QoS has been configured properly.

Since Receiver needs one decoder context for each remote surface it is receiving, and Transmitter needs one encoder context for each surface it is remoting, the CODEC must support multiple instances for encoding and decoding. However, hardware accelerated encoders and decoders may have very low limits on how many contexts they manage at a time. When there is no context available, remoting will fail.

4.3.4. Framerate and throttling

A Wayland application may attempt to update its window at arbitrary framerates which exceed the capability to encode, transmit, and decode frames. A similar case can also occur in a usual Wayland system where an application updates faster than the display can display, so the problem is not new and Wayland already accounts for it. The case is dealt by skipping or dropping frames and using always the latest one as possible. If the application stops updating, the latest committed frame is guaranteed to show up on display eventually.

Remoting adds more queues and latency to the output pipeline from application to display. Once a frame enters the encoder, it must be transmitted and decoded to avoid resetting the video stream. Frame dropping can occur before encoding, which similar to a usual Wayland compositor, and after decoding. Frame dropping after decoding means wasting the effort used for encoding and transmitting, therefore Receiver has to provide feedback on which frames have arrived and been displayed. This allows Transmitter to track the number of frames in, and the latency of, the encode-transmit-decode-display pipeline and throttle the application by stalling `wl_surface.frame` callbacks as needed. If the application keeps committing frames regardless, Transmitter will silently skip frames while the pipeline is too busy.

4.3.5. New GStreamer elements

A zero-copy capture element must be developed for bridging Weston with GStreamer in Transmitter. This is implemented in the Transmitter plugin and exposed to GStreamer as a GStreamer element.

It may also be necessary to develop or modify a GStreamer sink element for controlled display in Receiver. This will help synchronize the control messages in the control channel with frames being pushed to the destination compositor. Information about Receiver's commits to the destination compositor and the related feedback is needed back in Transmitter for throttling and feedback to the application.