Splitting GStreamer Pipelines

Jan Schmidt jan@centricular.com

A Coruña, 26th September 2023

Centricular

Monolithic Pipelines

- The original way
- Good for many scenarios
- Perfectly capable of dynamism - but the code is harder

Divide and conquer

Split pipelines into several smaller ones

Compartmentalize Code

- Modularity: easier to understand and maintain
- Different teams or people can work on (somewhat) self-contained pipelines
- Very dynamic pipelines can benefit from compartmentalization, f.ex., several hundred network clients coming and going every hour

Error Resilience

- Incoming video from camera
  - What if camera gets disconnected?
- Encode and write video to file
  - What if disk fails?
- Apply transforms, encode, write to network
  - What if the network goes down?
- None of these should bring down everything

Process Isolation

- Parsing of untrusted data
  - Demuxing/decoding of untrusted media
- Internet-facing interfaces
  - RTSP server, HTTP server, incoming RTP, etc.
- Actions that require elevated privileges
- DRM black-box

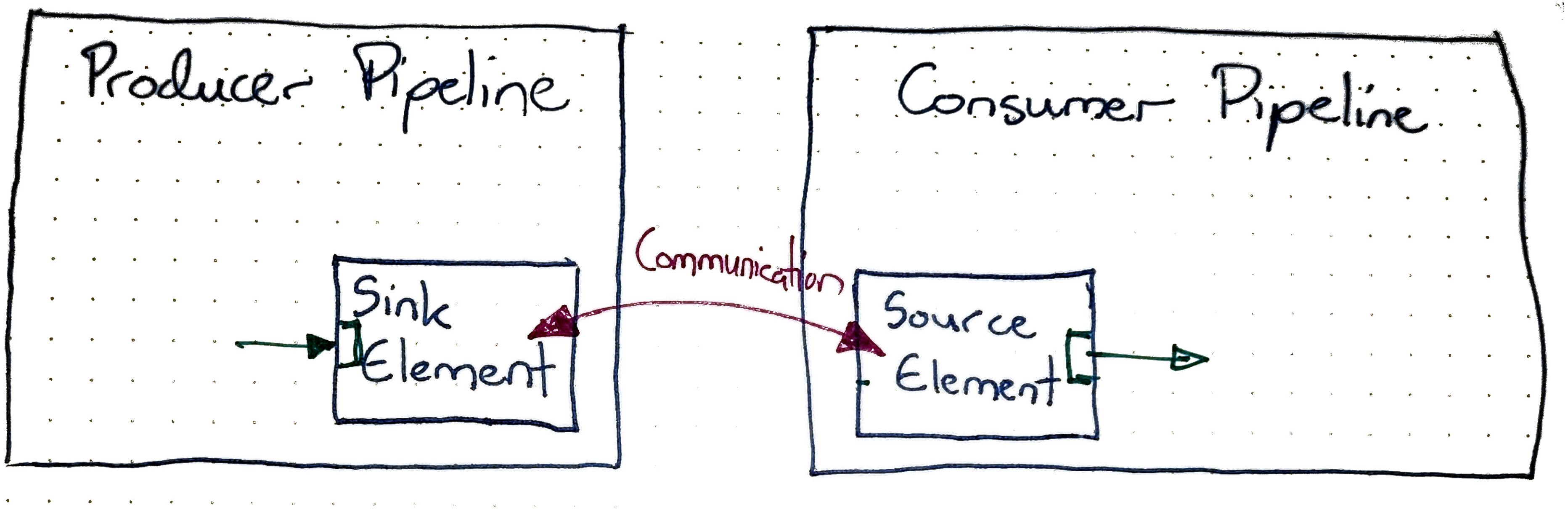
Easier Dynamism

- 1-to-N, one source to multiple sinks
- N-to-M, multiple sources to multiple sinks

GStreamer's Decoupling Mechanisms

So many approaches over the years

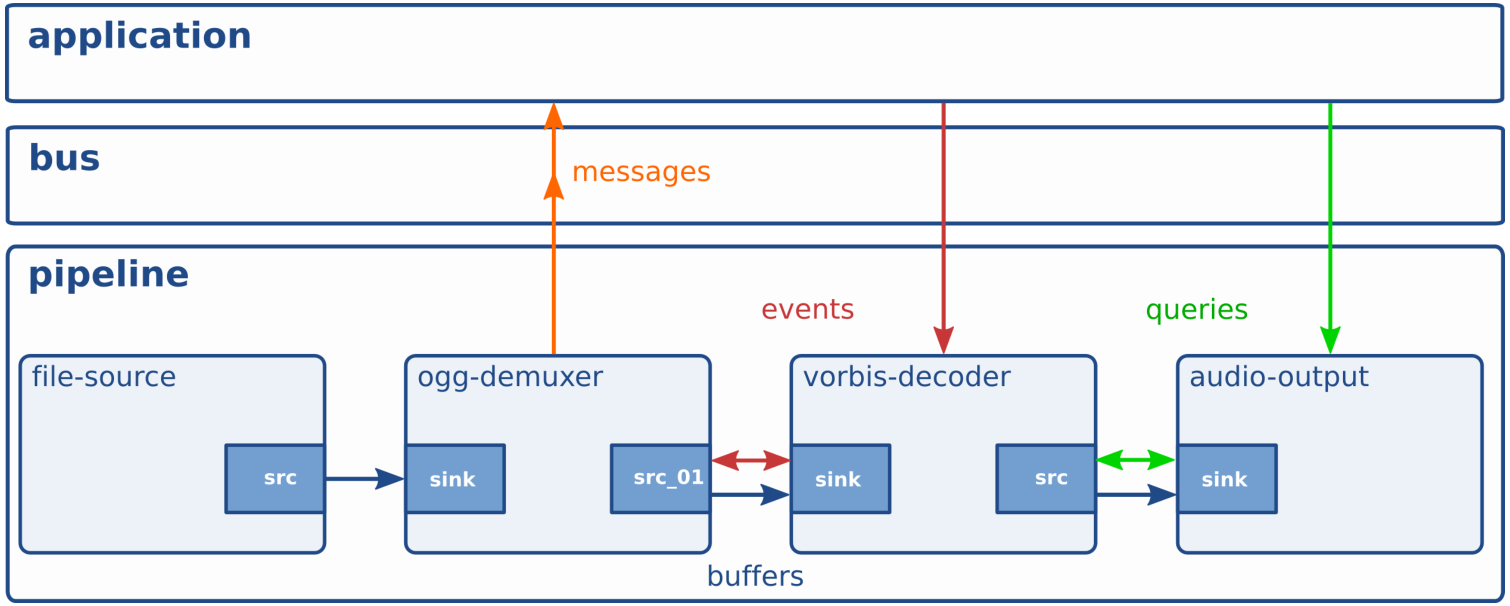
- Hard to even summarise in 40 minutes
- but one thing in common:



Centricular

What needs communicating?

- Exactly what depends on the problem



- but also, Bufferpools, Pipeline State, Clocks, GstContexts
- Different decoupling elements target different use cases

Decoupling Elements - Intra-process

- `appsink`, `appsrc`
- `proxysink`, `proxysrc`
- Original `inter*` elements (video, audio, subtitles)
- `gst-interpipes`
- New `inter` plugin

Decoupling Elements - Inter-process

- `shmsink`, `shmsrc`
- `ipcpipelinesink`, `ipcpipelinesrc`
- `cudaipcsink`, `cudaipcsrc`
- `unixfdsink`, `unixfdsrc` (in MR)
- Various network elements

Varied by linking method

- by code: `appsink` / `appsrc`
- by pointer: `proxysink` / `proxysrc`
- by channel string: classic `inter*`, `interpipes`, new `inter`
- by named pipe/unix domain socket: all the IPC elements

Format negotation

- Producer decides format: `appsink` / `appsrc`, classic `inter*`, new `inter`
- Upstream negotation: `proxysink` / `proxysrc`, `interipes`, `ipcpipeline`

Other query passing

- Queries are needed for bufferpool sharing or GstContext passing (intra-process)
- `proxysink` / `proxysrc` , `interpipes` , new `inter`
- `appsink` / `appsrc` can do allocation query in 1.24

1:1 vs 1:N data passing

- `proxysink` / `proxysrc` and `ipcpipeline` are 1:1
- Others all support 1:N

Zero copy

- Intra-process options are zero-copy - just passing buffers
- Inter-process: `shmsink` / `shmsrc` , `unixfd` elements can be

Queues / decoupling of receivers

- Internal queues (controllable size):
  - `appsink` / `appsrc`, `interpipes`, new `inter`, `cudaipc` (*)
- Internal queue (fixed size):
  - `proxysink` / `proxysrc`
- Direct connection (non-blocking):
  - classic `inter*` elements
- Direct connection (blocking):
  - `shmsink` / `shmsrc` (*)

Other notable features / differences

- `ipcpipeline` changes receiver pipeline state to follow the producer state
- `interpipe` elements adjust buffer timestamps for base time differences
- `inter` elements do latency queries properly for live pipelines
- No elements compensate for pipeline clock differences

PSA

- Watch out for `processing-deadline`!

Summary

	Link	nego	queries	1:N	Zero Copy	Buffering	IPC
`appsrc` / `appsink`	Code		*	X	X	X	
`proxysink` , `proxysrc`	Ptr	X	X		X	X	
original `inter*`	Name	X			X		
`gst-interpipes`	Name	X		X	X	X	
New `inter`	Name			X	X	X	
`shmsink` , `shmsrc`	Path			X	*	*	X
`ipcpipeline`	Path	X	X				X
`cudaipcsink` / `cudaipcsrc`	Path			X	X	X	X
`unixfdsink` / `unixfdsrc`	Path			X	*	?	X