

Let's solve our long standing issues

Time for GStreamer 2.0?

Thibault Saunier

GStreamer conference 2026 - Prague

Where this comes from

- We have been discussing 2.0 for a long time
 - what it would be about, what problems it should solve
- I started to implement the basic ideas we have been discussing
- Building the GStreamer of the future, with the lessons learned from 1.x

What needs a new major version

- Threading: elements own their threads
- C and the GLib at the heart of the API
- Negotiation, hard to debug
- Allocation in the data flow, little say for the app
- Dynamic pipelines
- Historical baggage we have to keep for API/ABI stability

Threading

- Elements spawn the threads: the application can't control how many

```
$ ges-launch-1.0 \  
  +clip a.webm start=0 duration=8.0 layer=0 \  
  +clip b.webm start=0 duration=8.0 layer=1 \  
  +clip c.webm start=0 duration=8.0 layer=2 \  
  +clip d.webm start=0 duration=8.0 layer=3
```

```
$ ls /proc/$PID/task | wc -l
```

49

11 matroskademux	5 GLib pools
9 multiqueue	4 queue

C and the GLib

- A great C foundation, ~25 years ago
- It is old, and today we can do better
- The GLib/GObject is not so welcoming
- The community is already moving to Rust
 - memory safety: we parse untrusted data all day
- Rust as the main language, behind a stable C API

Negotiation

```
$ gst-launch-1.0 videotestsrc \  
    ! video/x-raw, format=I420 ! x264enc \  
    ! video/x-h264, profile=high-10-intra \  
    ! fakesink  
ERROR: from element /GstPipeline:pipeline0/  
    GstVideoTestSrc:videotestsrc0:  
    Internal data stream error.  
streaming stopped, reason not-negotiated (-4)
```

- The error points at the source, not x264enc
- Templates can't express valid combinations
- Renegotiation is tricky, not transactional

Allocation

- In the data flow: it waits, or gets dropped
- Little say for the app over pools and memory
- Open questions:
 - could it be part of caps negotiation?
 - app provided buffers GStreamer has to use?

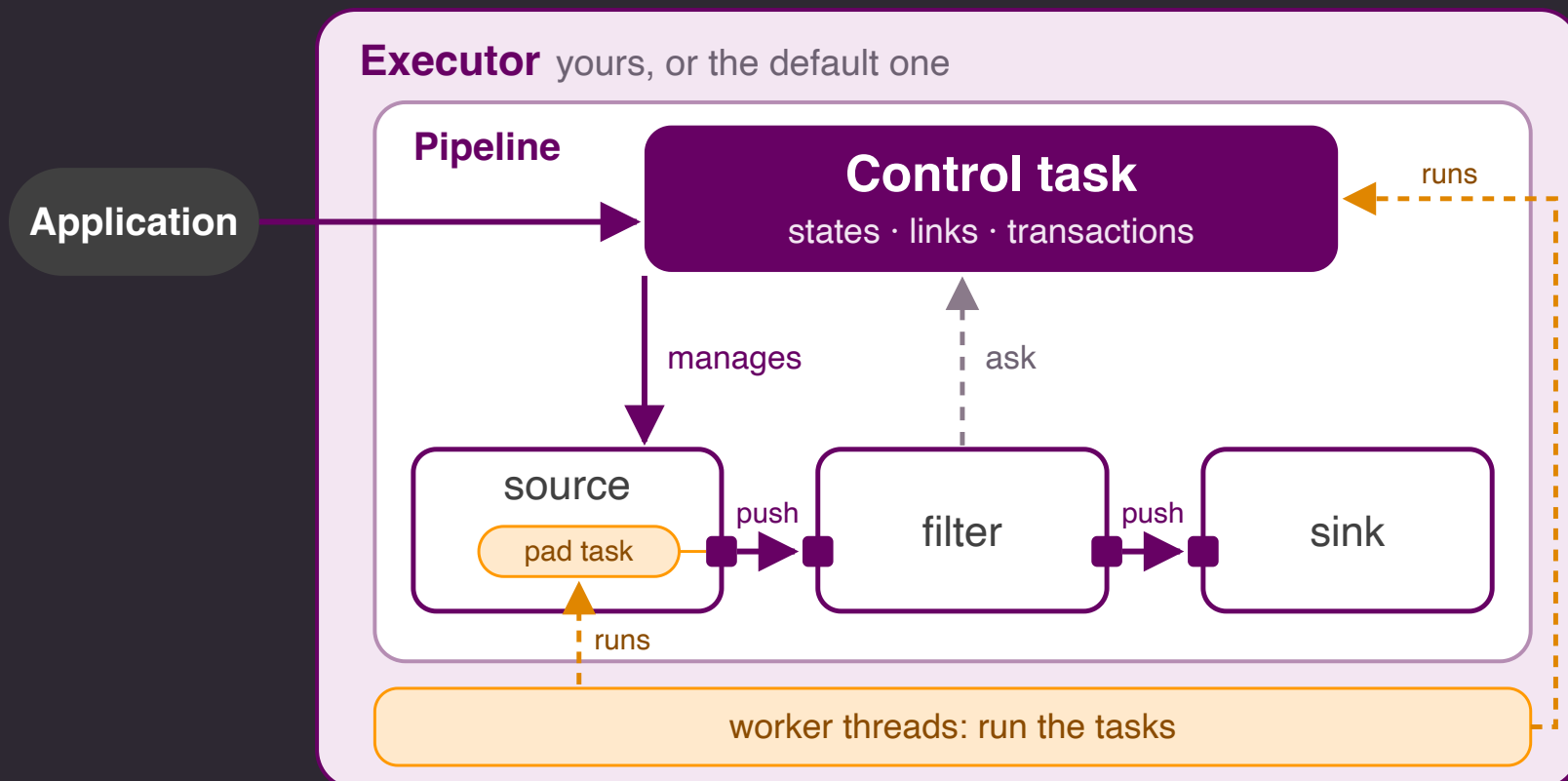
Dynamic pipelines

- Complex and error prone while data flows
- Easy to get wrong: races, deadlocks
- The trend is to split into pipelines to make everything simpler
 - linked with appsink / appsrc pairs
- Dynamic pipelines should be a first class citizen

gstreamer2

- <https://gitlab.freedesktop.org/thiblahute/gstreamer2>
- A very early stage core in Rust
- Try to remove as much complexity as possible from elements and apps
- Let the compiler check our rules

Scheduling: async first



Scheduling: your executor

- The core is executor agnostic and owns no threads
- A default executor for the standard use case
 - one thread per logical CPU core
- Fine-grained control of the scheduling for the app
- It plugs into the async runtime you already have
 - tokio, glib, the browser event loop in Wasm, a game engine loop, ...

The control task

- `set_state`, `add`, `remove`, `link`, `request pads`...
 - concurrent callers race each other
- One control task per pipeline owns them
- Every operation is an op, run to completion
- Elements can ask, not do

Dynamic pipeline: a transactional operation

- Stage a change: add, remove, relink, swap a branch
- Commit all at once, or roll back
- Prepare ahead, fire on a condition
 - stream time reached, EOS, error, a given clock time
- The streaming thread holds the data, the control task does the change

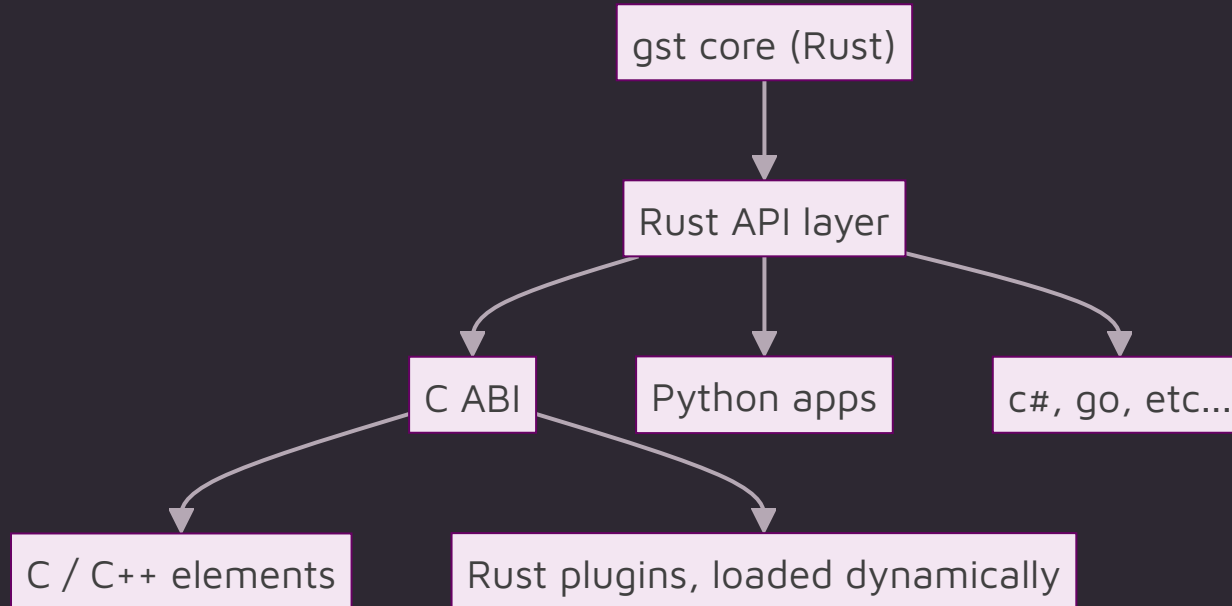
Where it stands

- caps, basic buffers, events and state changes
- Pads with push and pull mode streaming tasks
- executor, bins, pipelines
- First elements: fakesrc, identity, tee, fakesink
- Control task

Caps and values

- `ValueType` is a trait: plugin types get ranges and lists
- No empty caps: `Option<Caps>`
- A failed intersection says which field and why
- Fixated caps are their own type
- The negotiation experiments run on those types.

Bindings



- C to port elements, Python for application developers
- The goal: a stable C ABI, and packages for distros

Time for 2.0?

Thanks!

Any questions?

