PipeWire’s pipeline operation vs GStreamer’s explained

George Kiagiadakis
Principal Software Engineer

GStreamer Conference - October 2025

$ whoami

- Linux Systems Engineer with focus on multimedia
  - More specifically: Audio & Bluetooth
- Principal Software Engineer @ Collabora
- GStreamer, PipeWire & recently BlueZ
- Author & maintainer of WirePlumber
  - PipeWire's default session manager

COLLABORA

Outline

- Multimedia pipeline scheduling
- Scheduling in GStreamer
- Scheduling in PipeWire
- Combining approaches

COLLABORA

Multimedia Pipeline Scheduling

Pipeline scheduling - what is it?

- Managing when and how processing stages execute in a multimedia pipeline
- When: coordination and time sync with other stages
- How: what is going to call into the code to be executed? In parallel or in series?
- ***Goal***: ensure smooth, synchronized data flow

Pipeline scheduling - approaches

- When:
  - ASAP **vs** wait for event / timer
  - Wait to start **vs** wait to finish
- How:
  - chain-calling **vs** external loop
  - Wait internally **vs** externally
  - Single **vs** multi threaded

COLLABORA

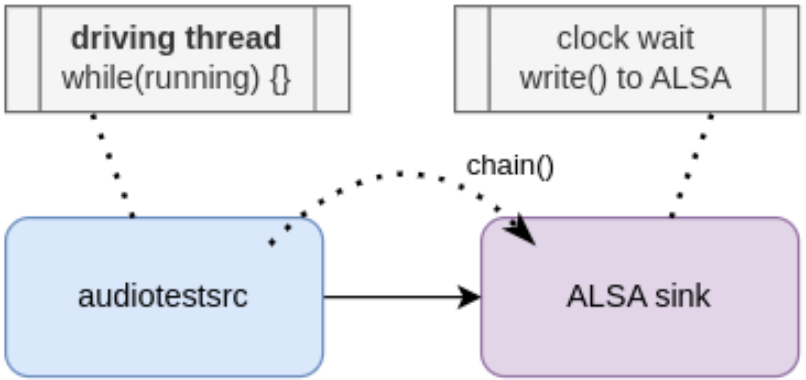
Scheduling in GStreamer

Scheduling in GStreamer

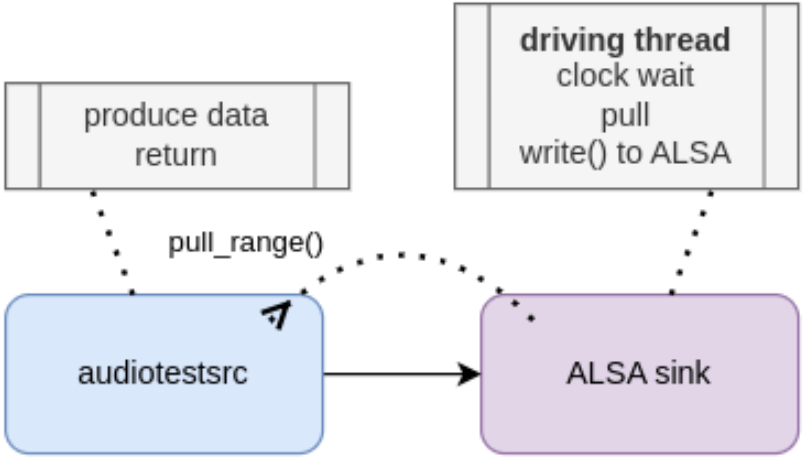
- Chain calling
  - Each element responsible for calling the next element
  - Push (source to sink) **vs** pull (sink to source) mode
- Multi-threaded
  - Each element responsible for its own thread(s)
- Wait internally
  - Each element responsible for waiting on the clock or device events
  - Multiple synchronization points possible
  - All start/finish waiting combinations possible

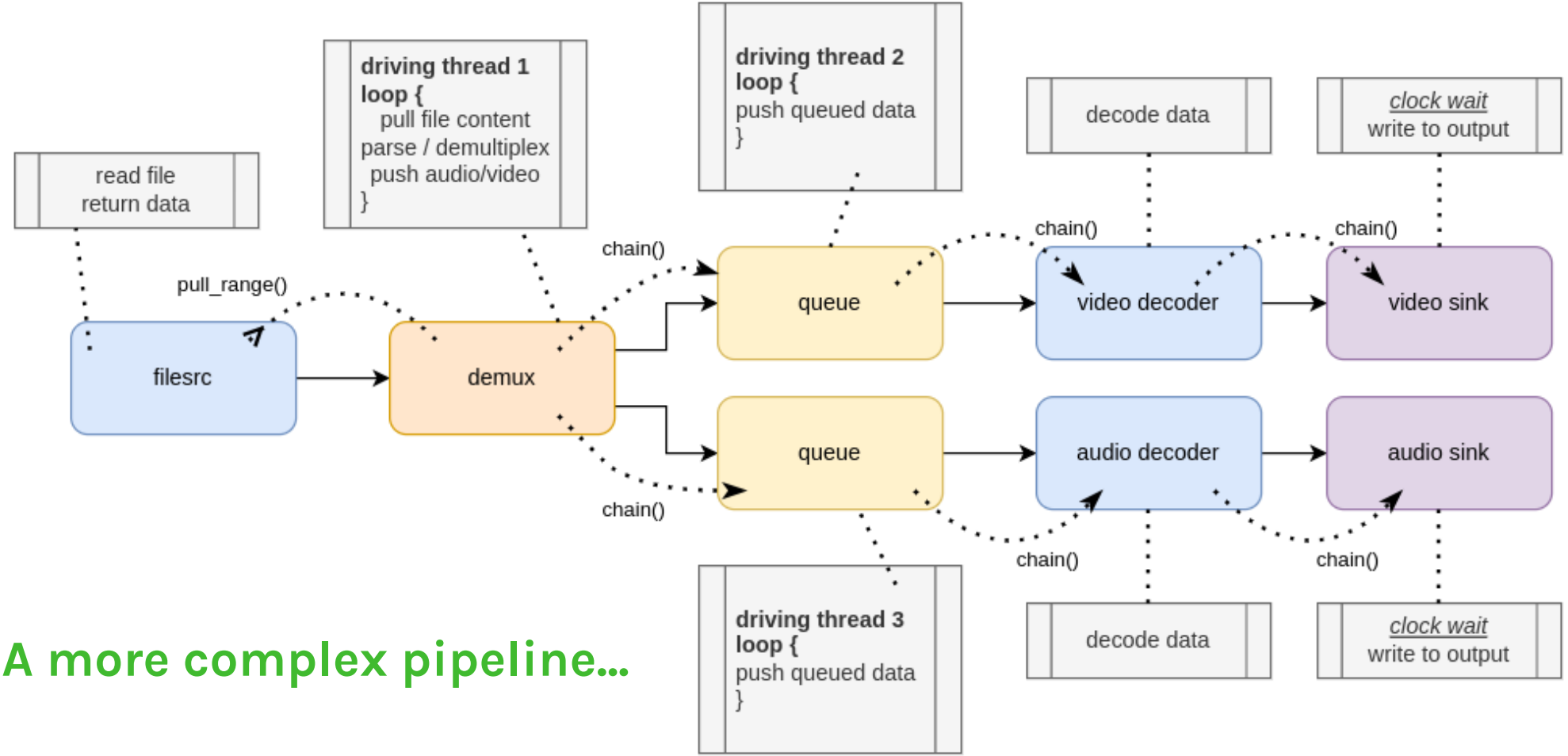
Scheduling in GStreamer

Push mode



Pull mode





A more complex pipeline…

COLLABORA

Scheduling in PipeWire

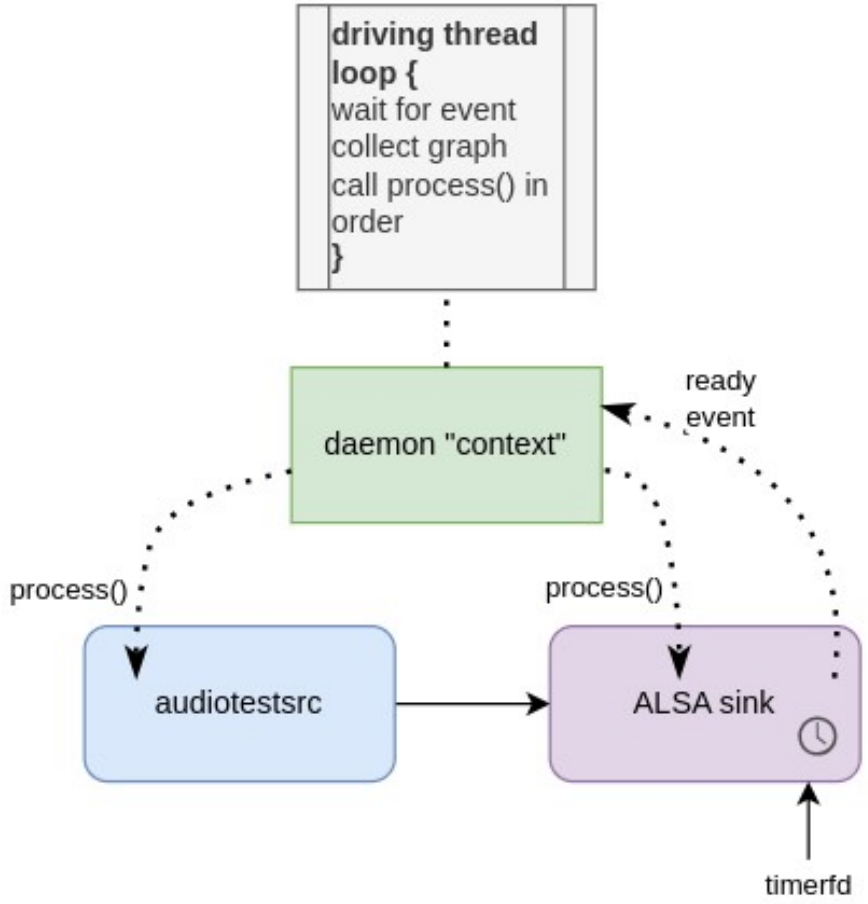
Scheduling in PipeWire

- Loop calling
  - Each element (“node”) responsible only for processing
- Single thread
  - Scheduling loop in a single thread
  - But processing may spin off to other threads or processes **
- Wait externally
  - One node designated as “driver” - signals the start of processing
  - Clock and processing are separate - no waiting inside the processing code

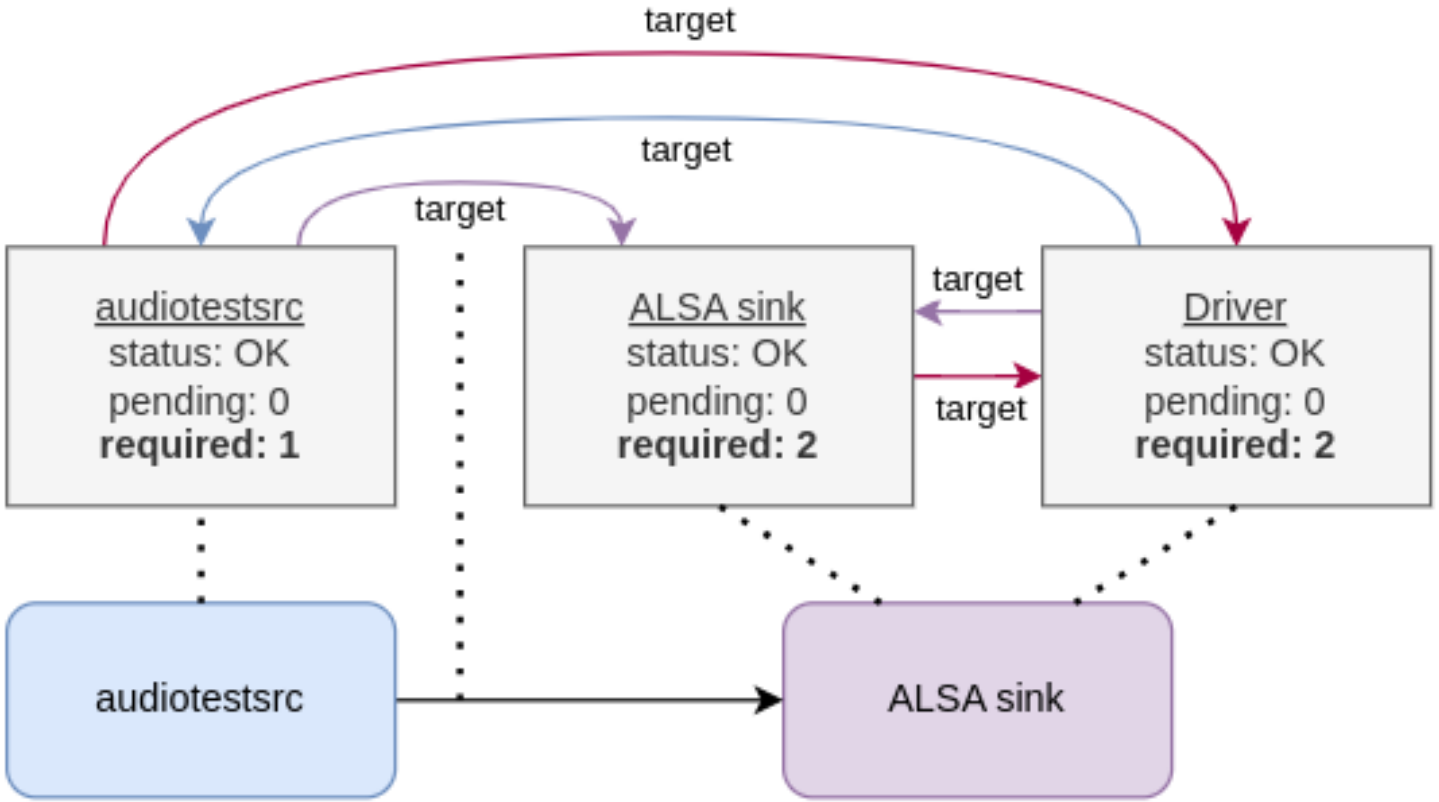
Scheduling: PipeWire vs GStreamer

	GStreamer	PipeWire
Calling into processing code	Chain	External loop
	Push/Pull	Only one way possible
Scheduling parallelism	Multi-threaded	Single-threaded
	Scheduling & processing together	Processing may use other threads
Waiting for events	Internal	External
	In line with processing	Clocking and processing are separate
	Multiple synchronization points	Single synchronization point

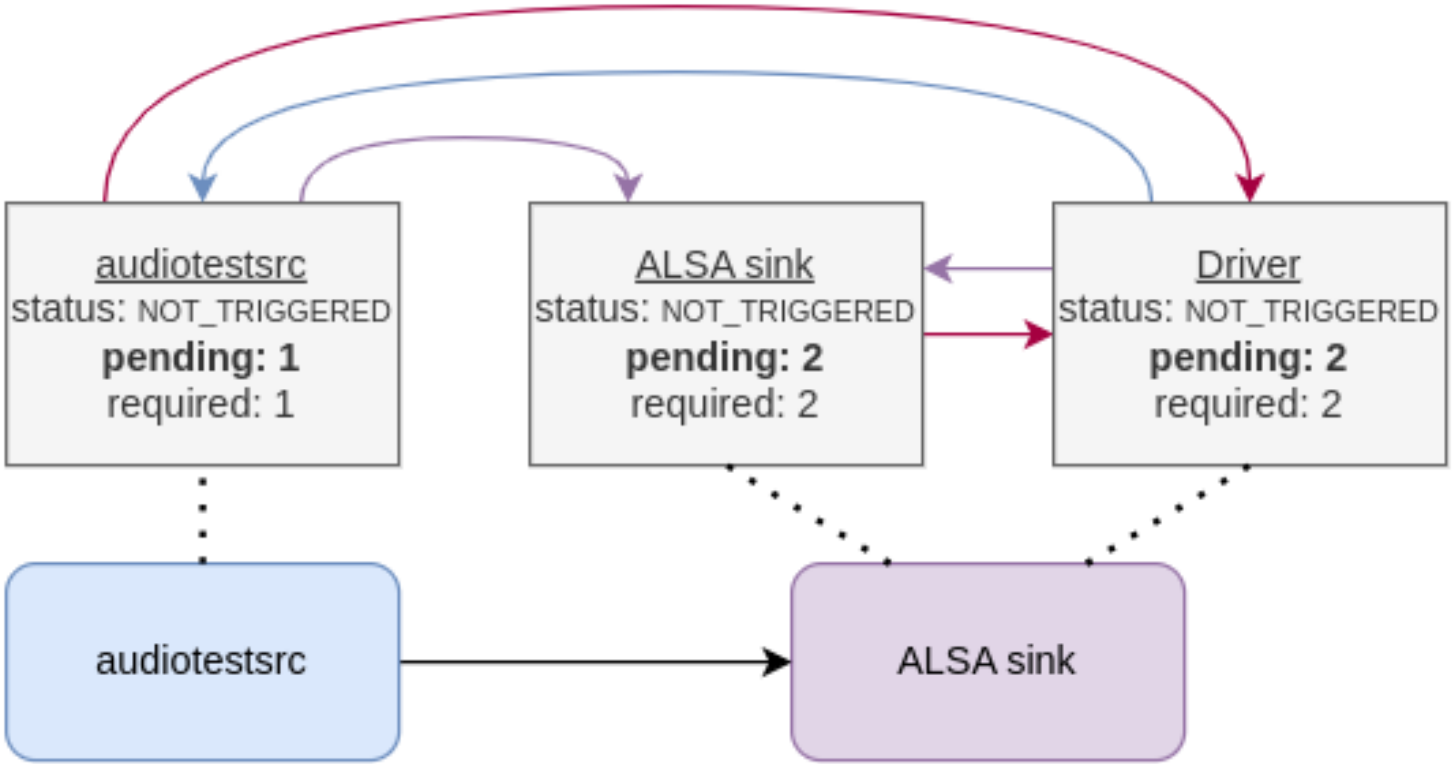
PipeWire graph



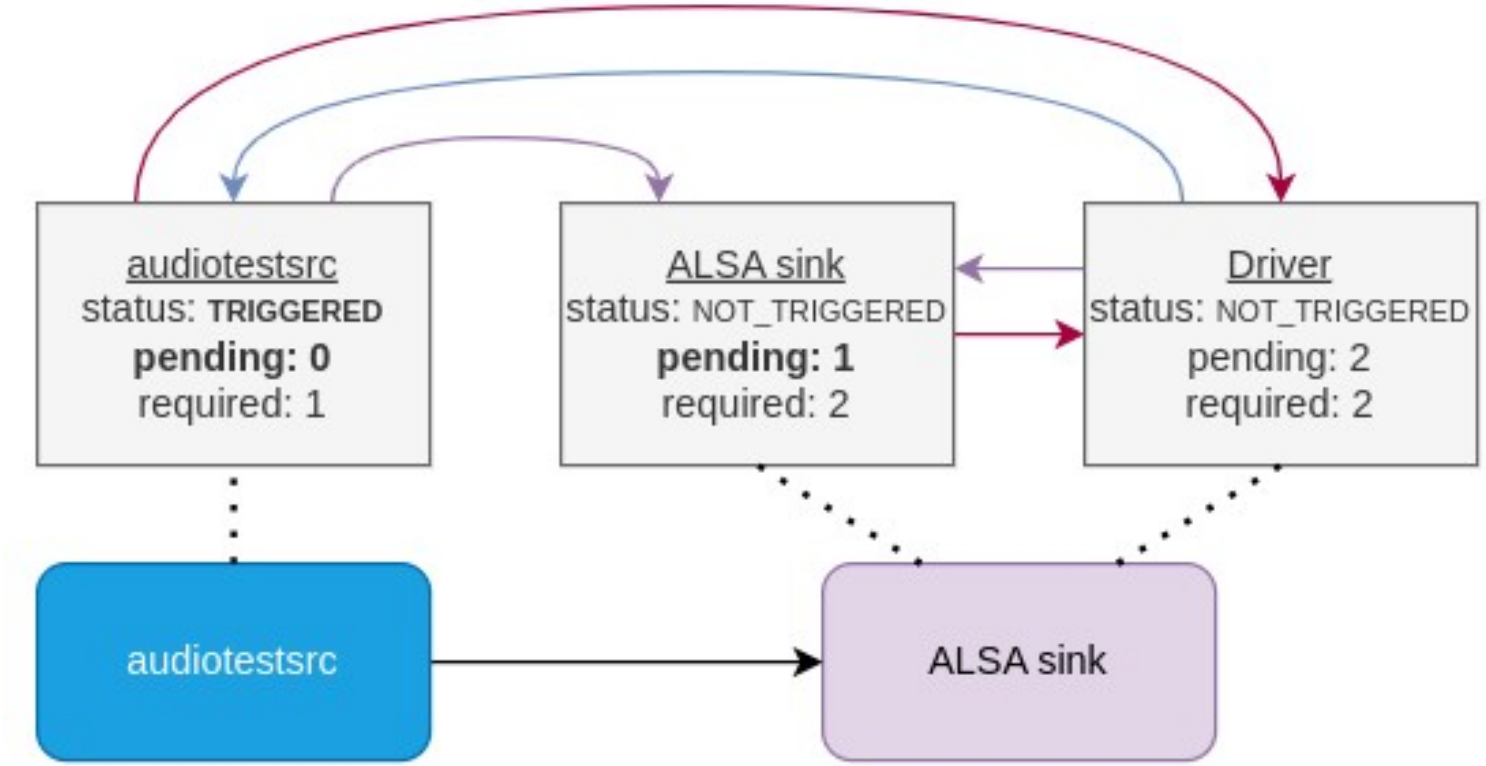
How does the loop know the order of execution?



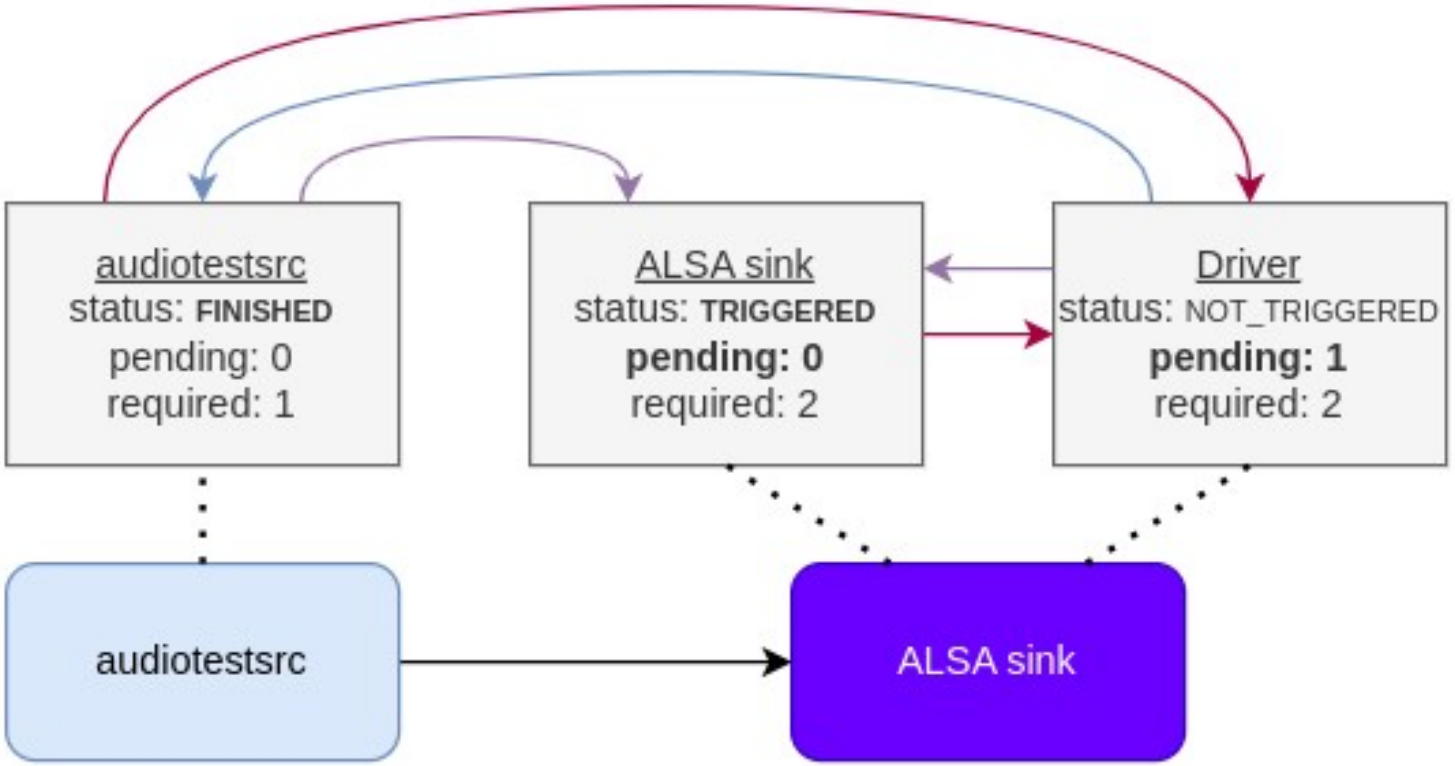
1/4 - Collect


audiotestsrc
status: NOT_TRIGGERED
pending: 1
required: 1
ALSA sink
status: NOT_TRIGGERED
pending: 2
required: 2
Driver
status: NOT_TRIGGERED
pending: 2
required: 2
audiotestsrc
ALSA sink

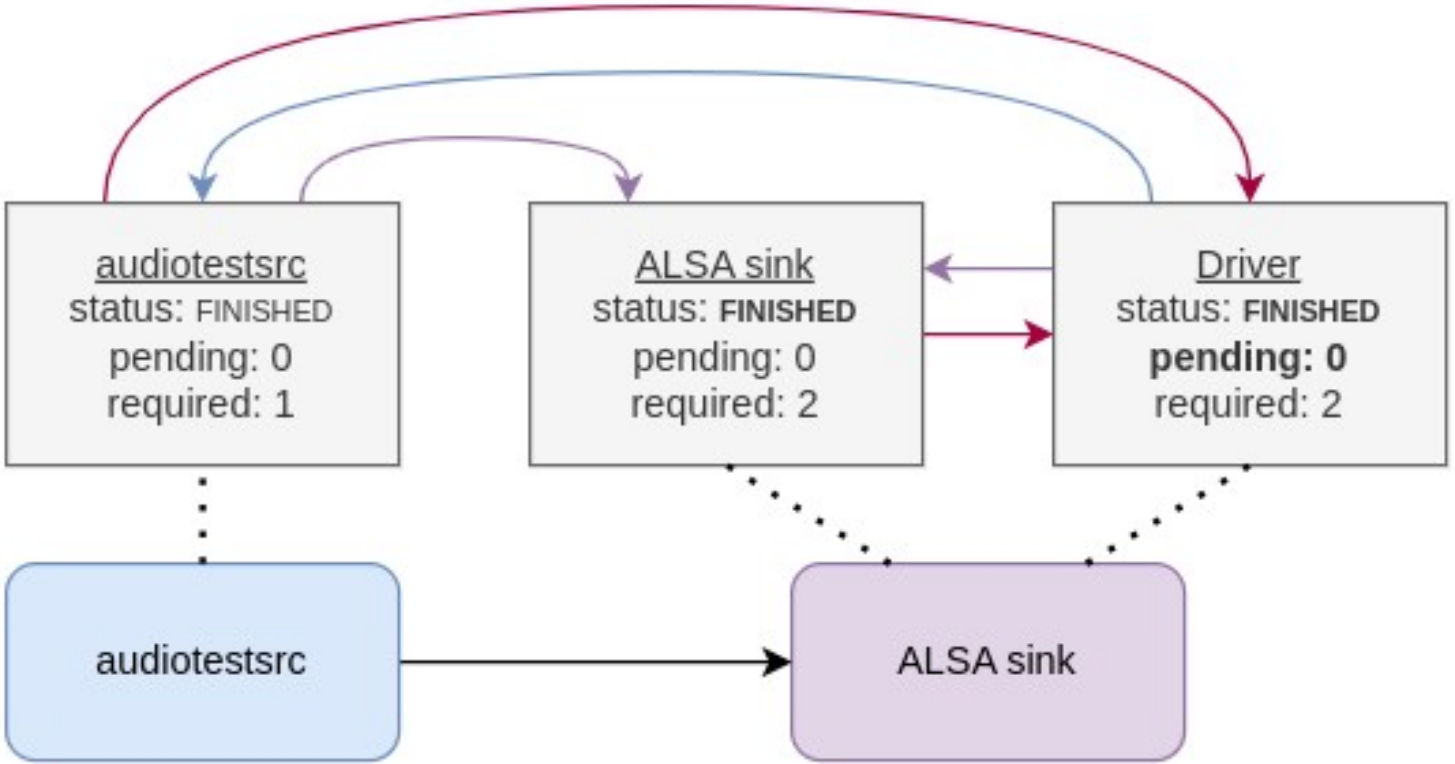
2/4 – Trigger audiotestsrc



3/4 – Trigger ALSA sink



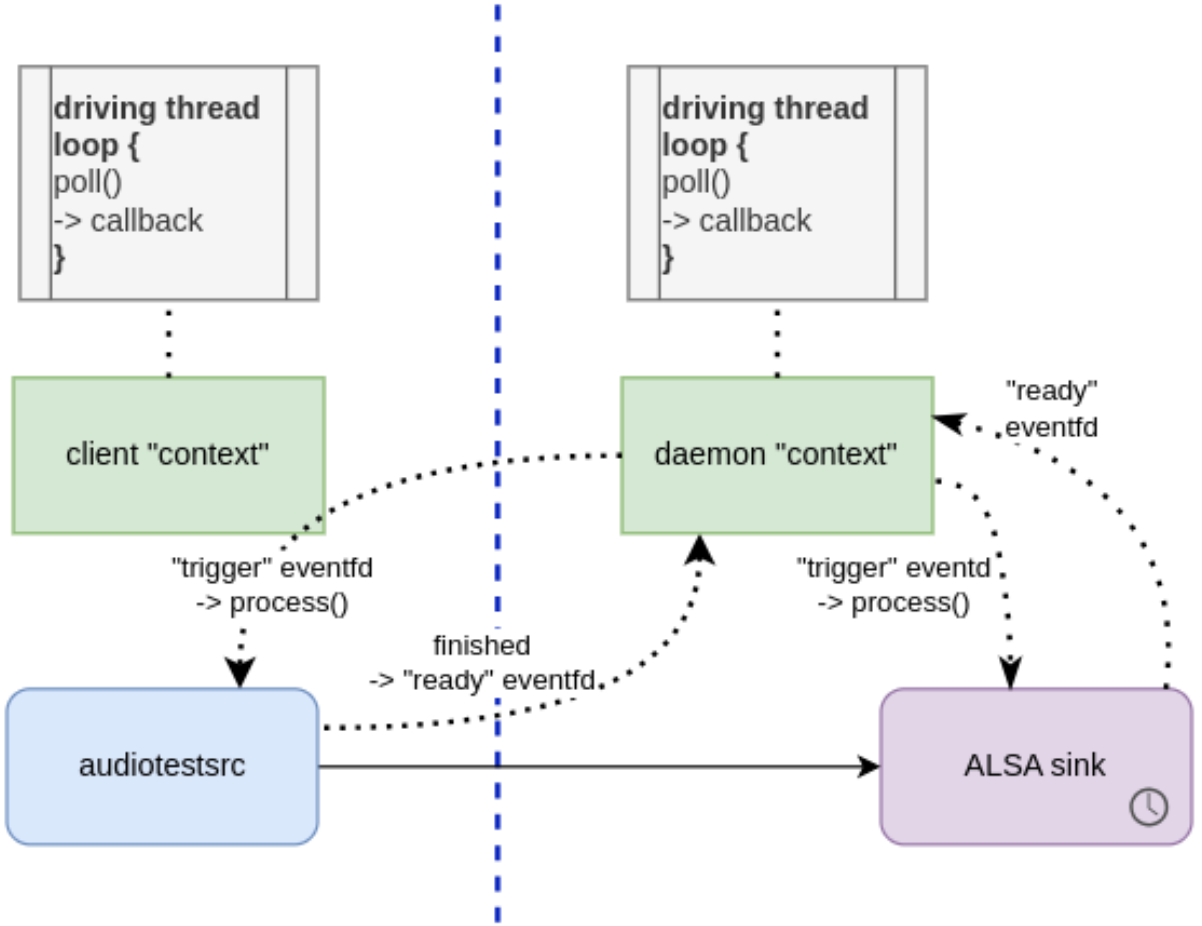
4/4 – Finish


audiotestsrc
status: FINISHED
pending: 0
required: 1
ALSA sink
status: FINISHED
pending: 0
required: 2
Driver
status: FINISHED
pending: 0
required: 2
audiotestsrc
ALSA sink

Signaling

- Triggering a node uses an **eventfd**
  - `process()` function is a callback to that eventfd
- “Ready” event also uses an **eventfd**
  - Driving “loop” is a stateful callback; ready event used again to return execution
- Driver wakeup uses a **timerfd** (or some device fd)
- Driving thread `poll()`s for all of them
  - Callback execution multiplexed within the same thread

Multi-process !



Data sharing for multi-process

- Nearly all node structures live on **memfd** shared memory
  - Even in single process scenarios
- Atomic writes and protocol to ensure integrity
  - No locks in the driving thread ! – It must *never* block.
- Data passing works through fixed-size arrays of buffers
  - Separate structure to communicate which buffer was written in this cycle

Cycle duration

- Nodes **must** complete processing before next wakeup event
  - Predictable and (almost) constant cycle duration
  - Note: driving thread has real-time priority
- What if a node takes too long to process?
  - If on separate process, it can be **ignored** – output data filled with zeros
    - Missing data from some path, but the rest of the graph can still run
  - If on the same process, it **blocks** the driving thread (too bad !!!)
    - Driver underruns, nothing can finish
    - If it takes way too long, kernel will SIGKILL the thread (because it's RT)

COLLABORA

Combining approaches

Lessons learned

- GStreamer approach is very flexible
  - Both live and non-live pipelines
  - But no real-time guarantees!
  - Elements may block execution as needed: allocations, locks, blocking I/O, etc…
  - Element processing time is not considered as latency
  - Data rate throttles as needed to get things done

Lessons learned

- PipeWire approach can guarantee real-time
  - Very fast execution, can do <1ms cycles
  - But is not as flexible...
  - Meant for live pipelines only

Combining approaches

- PipeWire + GStreamer with pipewiresrc/pipewiresink
  - You need to understand what you are doing
  - GStreamer pipeline must be able to respect timing and buffer management constraints
- GStreamer threadshare elements
  - PipeWire's scheduling approach within GStreamer
  - Meant for elements that do a lot of I/O
  - Still not real-time, due to GStreamer's design

Thank you!