audio.dev CONFERENCE BRISTOL & ONLINE 2026

Functional Reactive Programming for Audio Software

What a Real-Time Concurrent FRP Engine Can Do for Us

Eloi Marín Gratacós

audio.dev CONFERENCE BRISTOL & ONLINE 2026

Concurrent Synchronous

Reactive Programming for Audio Software

What a Real-Time Concurrent CSRP Engine Can Do for Us

Eloi Marín Gratacós

audio.dev CONFERENCE BRISTOL & ONLINE 2026

Concurrent Synchronous Reactive Programming for Audio Software

What a Real-Time Concurrent CSRP Engine Can Do for Us

Eloi Marín Gratacós

** I used LLMs both for the library's implementation and the presentation slides*

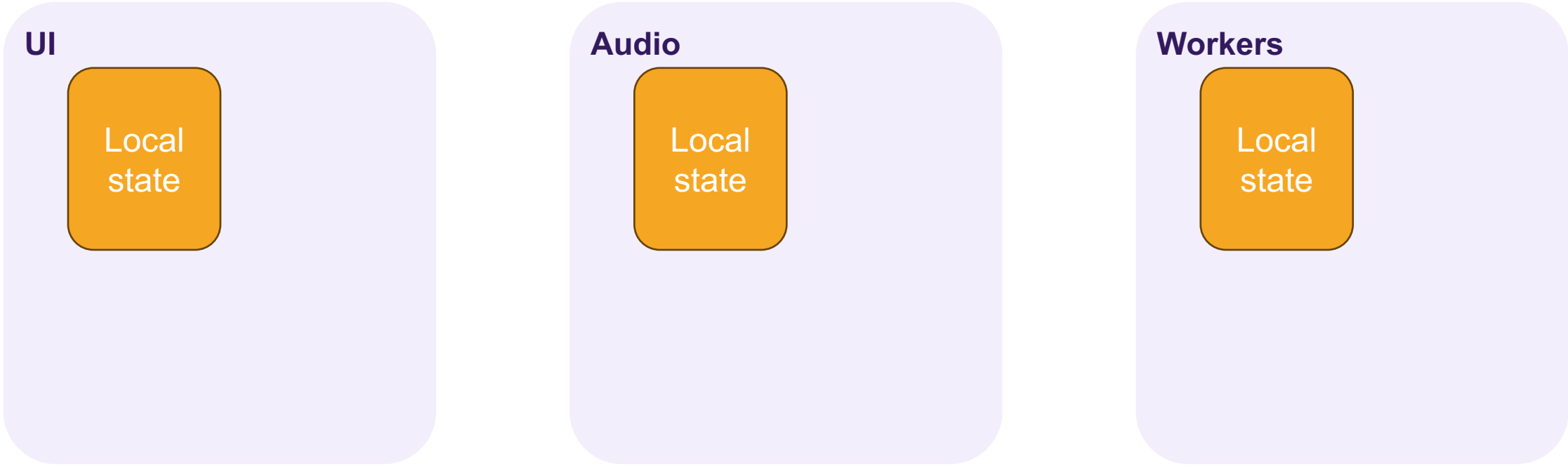
Concurrent clocks for real-time-safe logic

UI

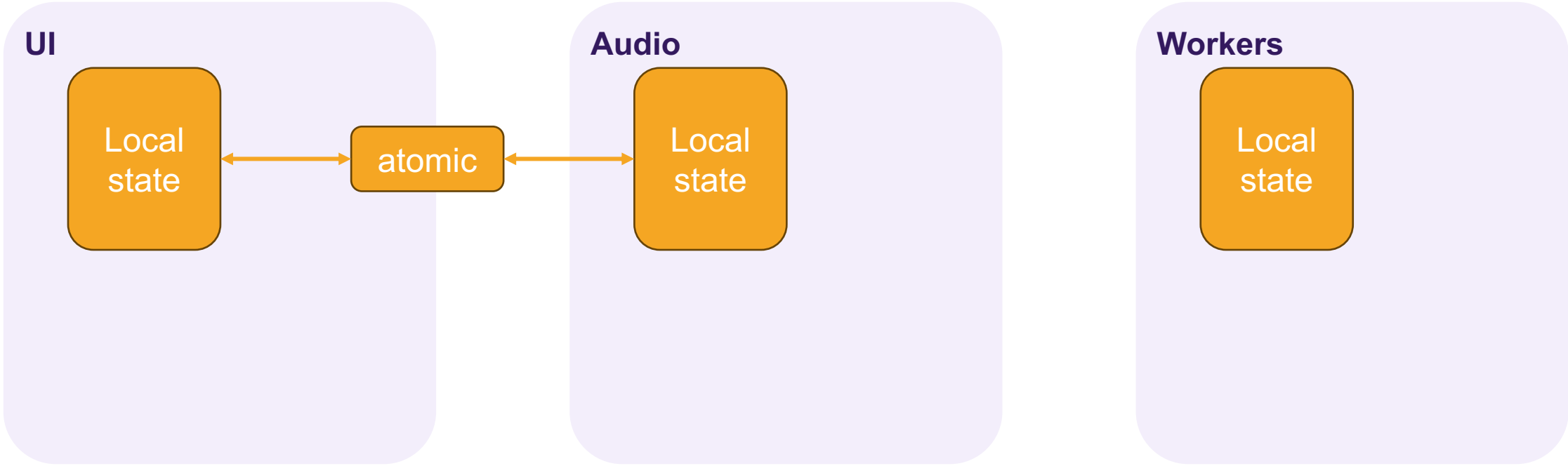
Audio

Workers

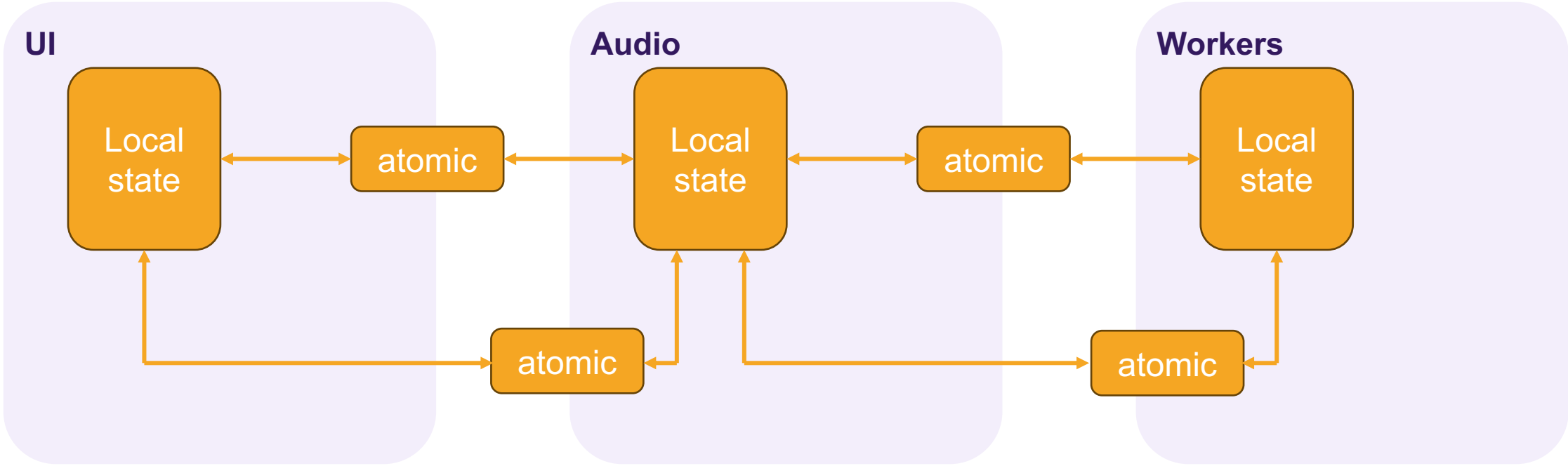
Concurrent clocks for real-time-safe logic



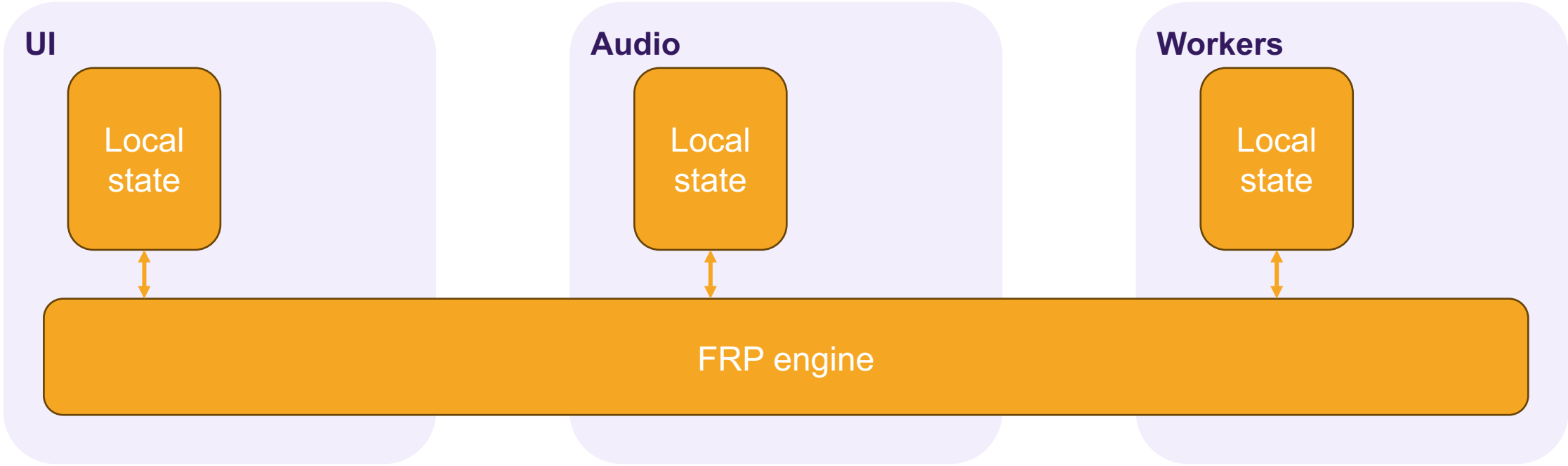
Concurrent clocks for real-time-safe logic



Concurrent clocks for real-time-safe logic



Concurrent clocks for real-time-safe logic



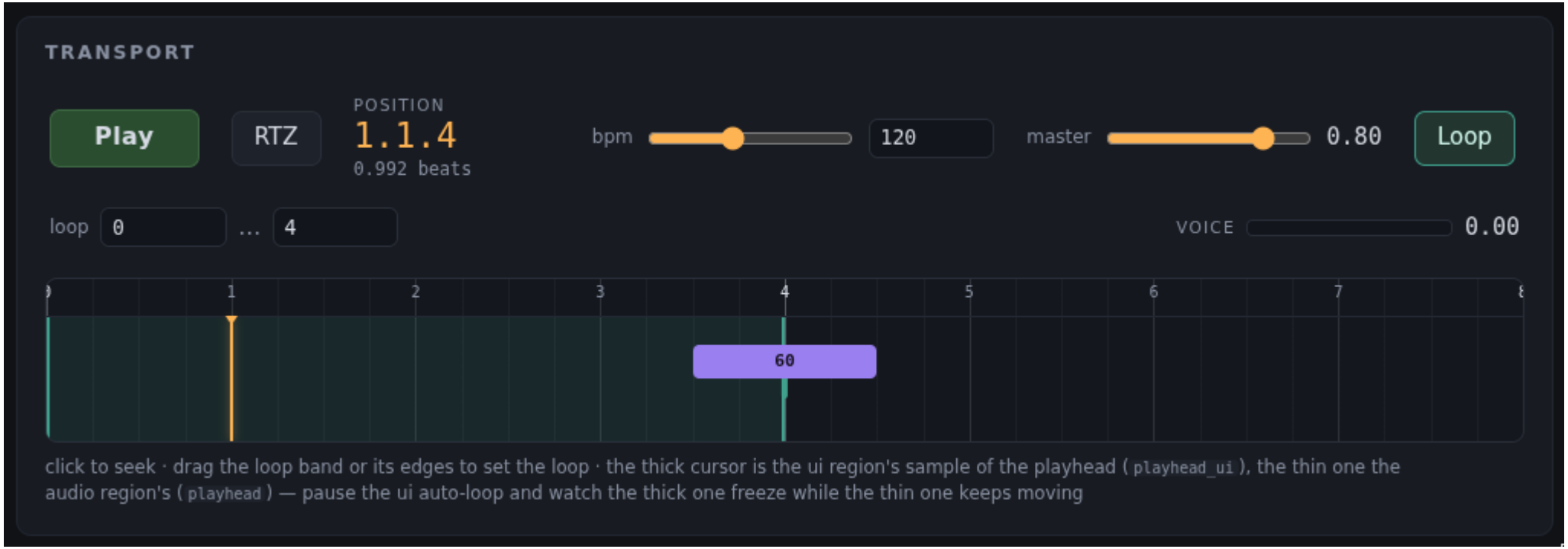
PART 1 · DEMO

The DAWmulator

“You shouldn't write a DAW”, so let's do it!

youtube.com/watch?v=GMInh6_9aTc ("Why You Shouldn’t Write a DAW - David Rowland – ADC23)

DAWmulator


TRANSPORT
Play
RTZ
POSITION
1.1.4
0.992 beats
bpm
120
master
0.80
Loop
loop
0
...
4
VOICE
0.00
1
2
3
4
5
6
7
60
click to seek · drag the loop band or its edges to set the loop · the thick cursor is the ui region's sample of the playhead (playhead_ui), the thin one the audio region's (playhead) — pause the ui auto-loop and watch the thick one freeze while the thin one keeps moving

Playhead in FRP

UI

Audio

Playhead in FRP



Playhead in FRP

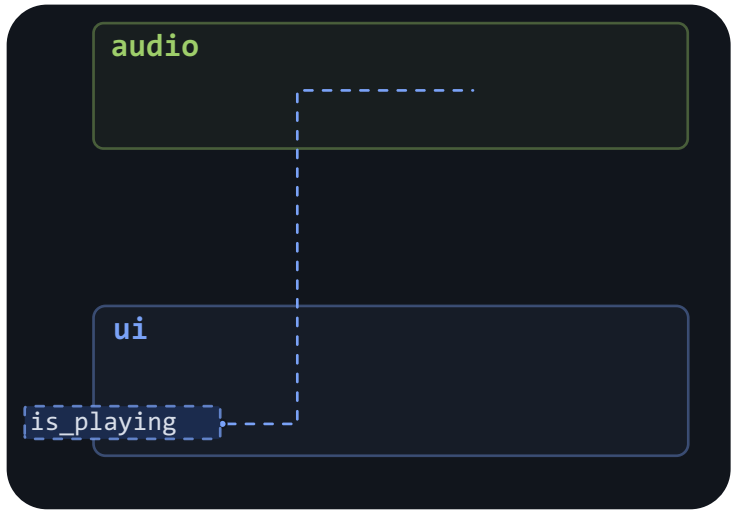


API sneak peak...

operator	what it does
`input/output`	Ports: They declare how data enters and leaves the engine
`at`	Declares crossings to another region, with a policy: latest or queued
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...

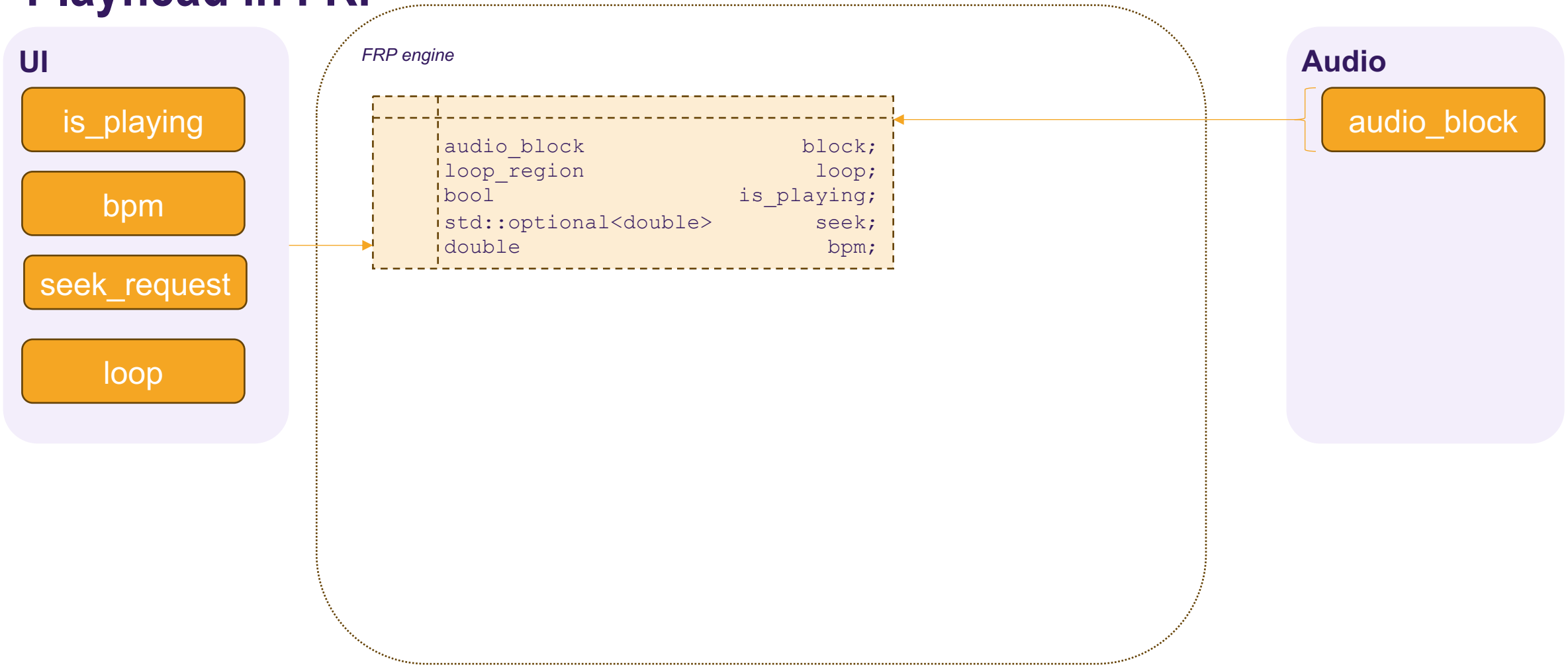
Operators `input` and `at`

```
at<audio, latest>(
    input<boolean, ui>("is_playing")
)
```



latest	The most recent value as of the consumer's tick.
queued	Every value, in order, up to the size of its buffer.

Playhead in FRP

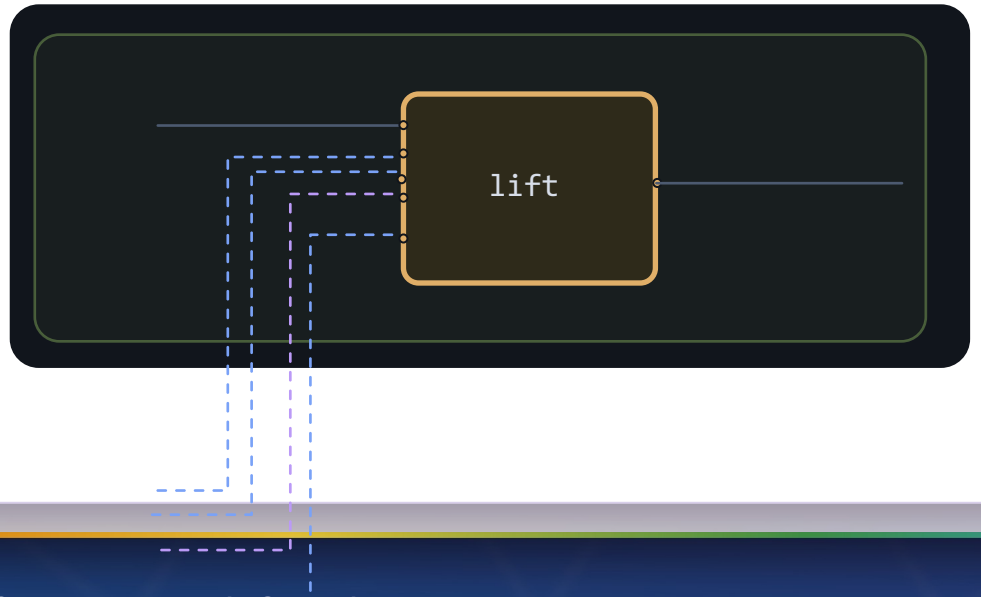


API sneak peak... (2)

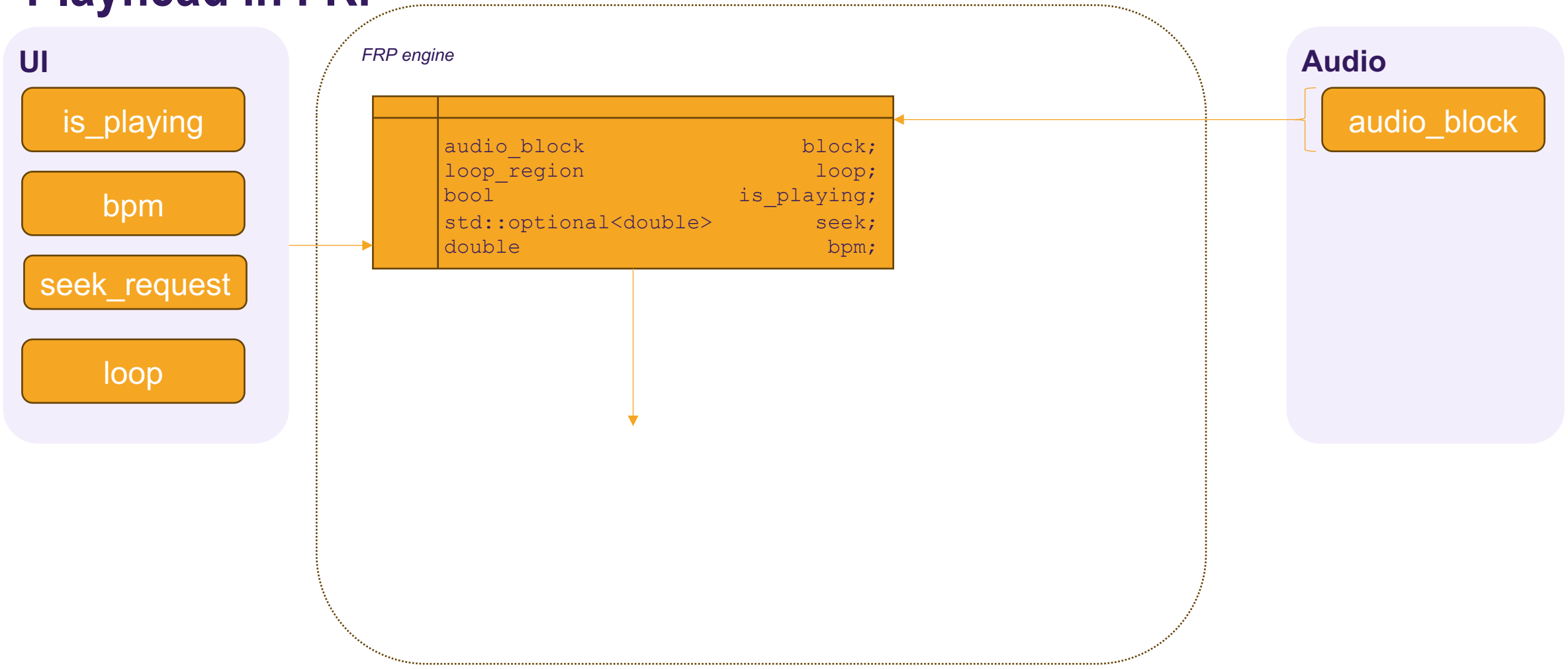
operator	what it does
`input/output`	Ports: They declare how data enters and leaves the engine
`at`	Declares crossings to another region, with a policy: latest or queued
`lift`	Applies a pure function
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...

Operator `lift`

```
lift<input_t, output_t>(
    [](const t1& p1, const t2& p2, const t3& p3, const t4& p4, const t5& p5)
    {
        return output_t{ p1, p2, p3, p4, p5 };
    }
)
```



Playhead in FRP



API sneak peak... (3)

operator	what it does
`input/output`	Ports: They declare how data enters and leaves the engine
`at`	Declares crossings to another region, with a policy: latest or queued
`lift`	Applies a pure function
`scan`	State over time: s(t) = f(s(t−1), in(t))
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...
`...`	...

Operator `scan`

```
scan<input_t, double>(
    0.0, // Initial state
    [](double current_pos, const input_t& input)
    {
        return playhead_step(current_pos, input);
    }
)
```

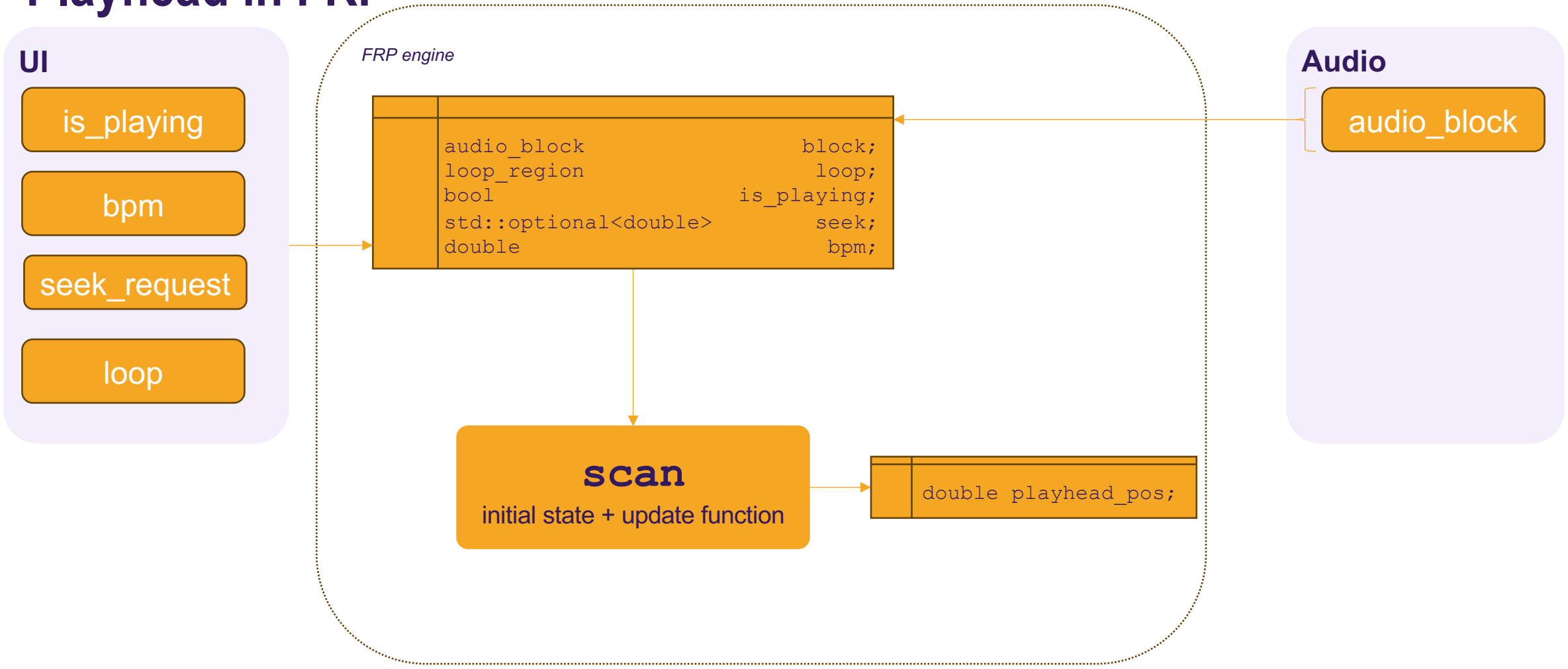
One playhead step

```
constexpr
double
playhead_step(double current, const input_t& in)
{
    if (in.seek)
        return wrap_pos(in.seek, in.loop);

    if (!in.playing)
        return current;

    return wrap_pos(
        current + beats_per_block(frame_count, sr, bpm),
        in.loop
    );
}
```

Playhead in FRP



PAUSE

Questions until here?

Next: a bit of theory, then the actual C++ library.

PART 2 · THEORY

What FRP really is

Conal Elliott and Paul Hudak, “Functional Reactive Animation”, 1997

PA

W

Cor

A confession

PAI

W

Cor



A confession

PAI

W

Cor





A confession

PAI

W

Cor





A confession

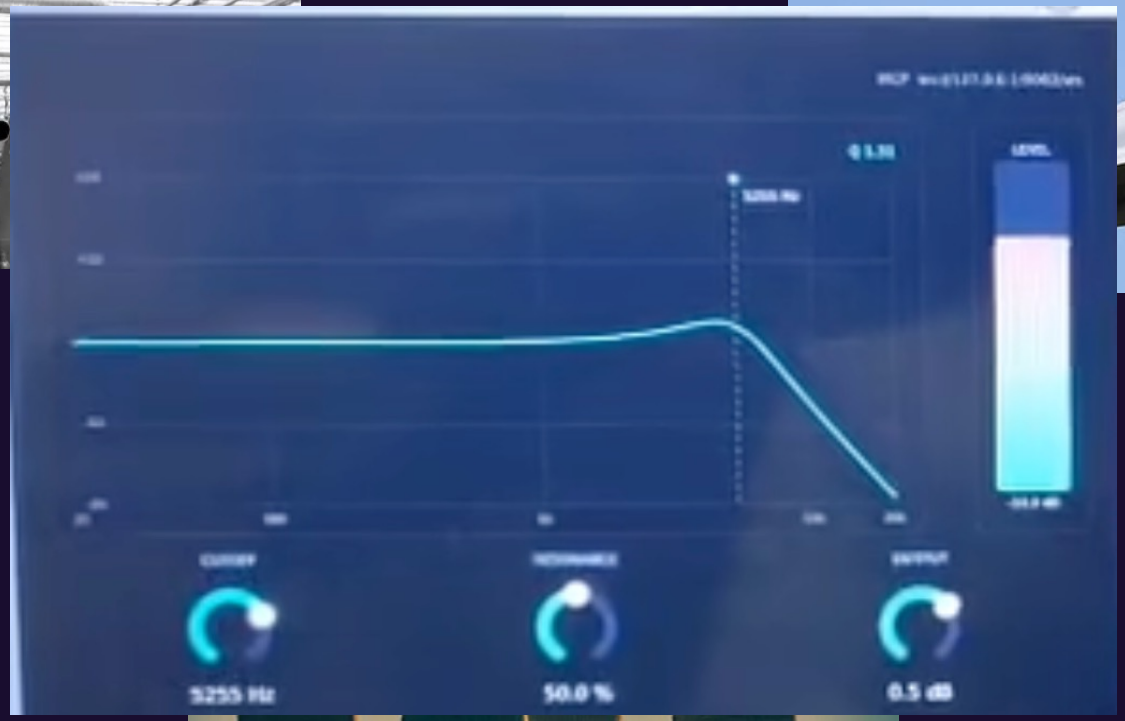


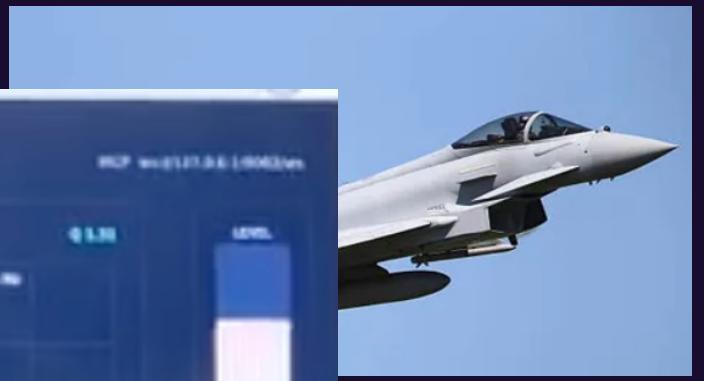
PAI

W

Cor









PART 3 · THE LIBRARY

Meet clockr!

The whole vocabulary

operator	what it does	in the DAWmulator
`input / output`	Ports: where values enter and leave a region	playing, seek… / playhead
`at`	Crosses to another region, with a policy: latest or queued	every UI → audio value
`lift`	Applies a pure function (map)	building TransportIn
`scan`	State over time: s(t) = f(s(t−1), in(t))	the playhead
`delay`	Unit delay, seeded with an initial value	—
`zip`	Fan-in: several inputs into one lift	the five transport inputs
`share`	Fan-out: one node, several consumers	bpm, loop, seek
`parallel`	Groups independent chains	the six outputs
`>>`	Composes: the output of one into the next	everywhere
`stepper`	Sugar: keeps an event's last value	—

The playhead code

```
share(
    zip(
        input<BlockDesc, clock_tag::audio>("block"),
        at<clock_tag::audio, conc::policy::latest>(input<bool, clock_tag::ui>("is_playing")),
        at<clock_tag::audio, conc::policy::latest>(input<LoopRegion, clock_tag::ui>("loop")),
        at<clock_tag::audio, conc::policy::queued>(input<std::optional<double>, clock_tag::ui>("seek")),
        at<clock_tag::audio, conc::policy::latest>(input<double, clock_tag::ui>("bpm"))
    )
    >> lift<transport_in_wire, transport_in>(
           [](const BlockDesc& b, const bool& p, const LoopRegion& l, const std::optional<double>& sk, const double& bpm_v)
           {
               return transport_in
               {
                   .block = b,
                   .is_playing = p,
                   .loop = l,
                   .seek = sk,
                   .bpm = bpm_v,
               };
           })
    >> scan<transport_in, PosPair>(
        PosPair { 0.0, 0.0 },
        [](const PosPair& state, const transport_in& in)
        {
            return PosPair { state.cur, daw::transport_step(state.cur, in) };
        }
    )
)
```

The playhead network



The debugger



One clock tick

1. **Read**
2. **Evaluate**
3. **Settle**
4. **Publish**



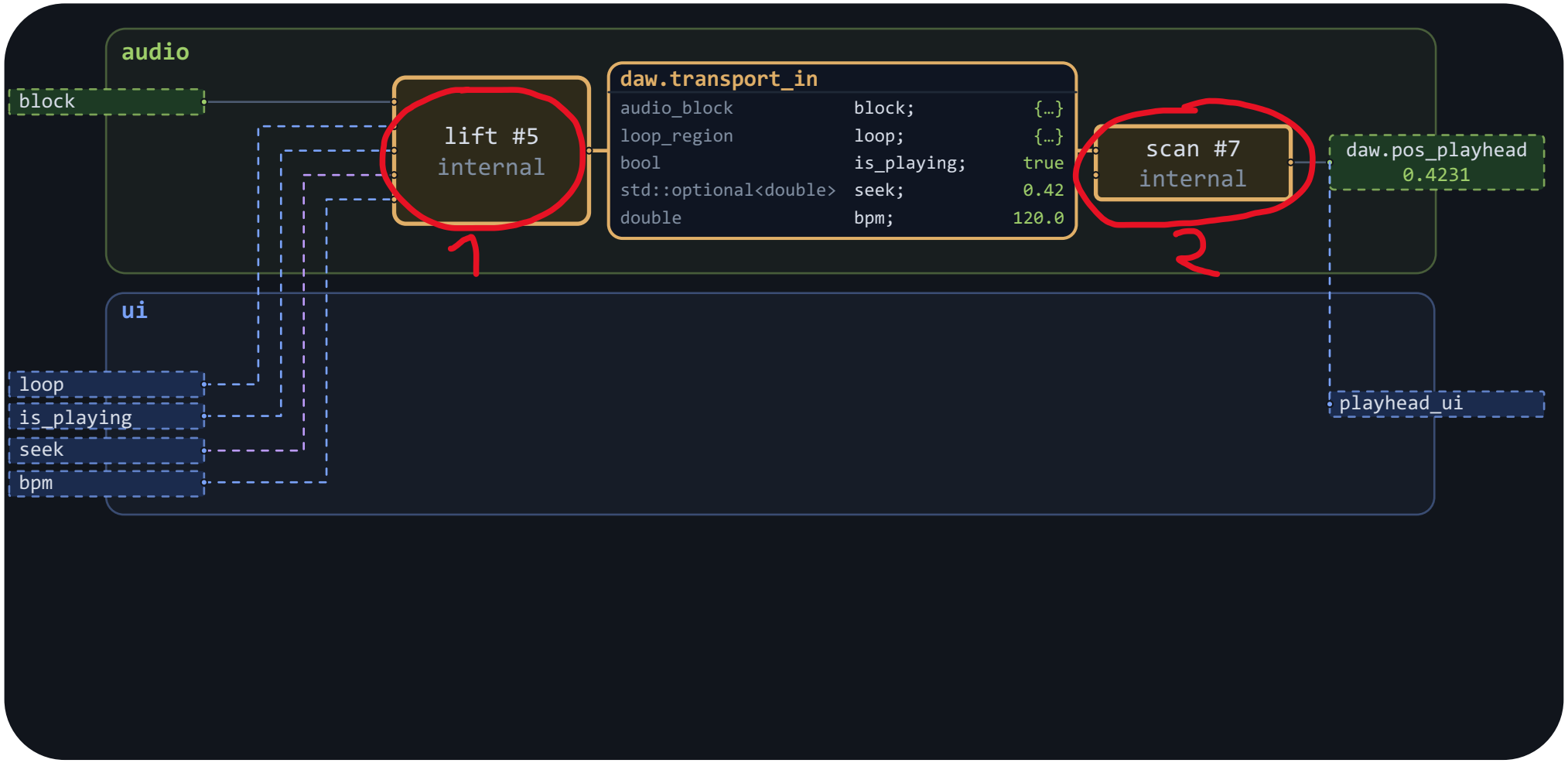
One clock tick

1. **Read**
2. **Evaluate**
3. **Settle**
4. **Publish**



One clock tick

1. **Read**
2. **Evaluate**
3. **Settle**
4. **Publish**



One clock tick

1. **Read**
2. **Evaluate**
3. **Settle**
4. **Publish**

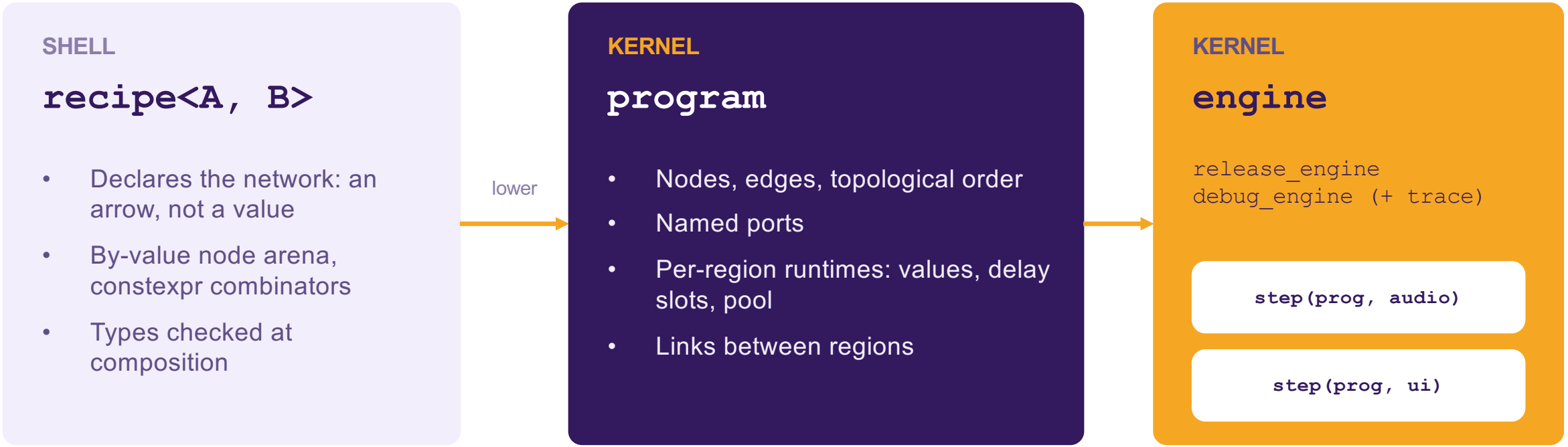


One clock tick

1. **Read**
2. **Evaluate**
3. **Settle**
4. **Publish**

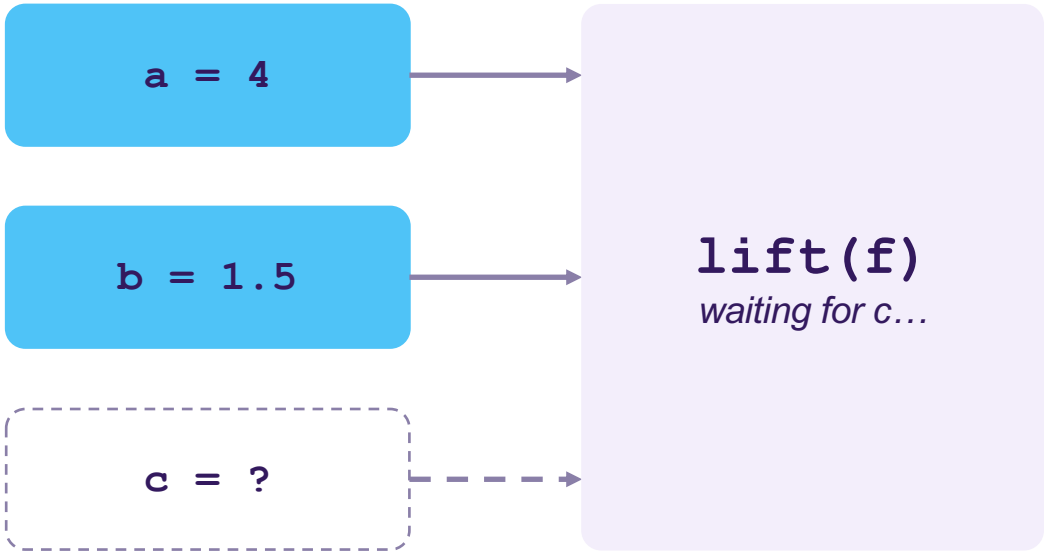


Two planes: shell and kernel



Lowered at compile time (compile) or at startup (program{recipe}). Either way the recipe can die: only the topological indices remain.

Why value lifetime is hard in push-based FRP



a and b must live somewhere until c arrives.

Lifetime is not deterministic: it depends on the state of the rest of the graph.

- lift fires only once all its inputs have values
- Values that already arrived wait for their siblings
- When those arrive depends on arbitrary upstream logic

In clockr every input has a value on every tick: nothing waits. What remains: values crossing threads.

Values in the kernel: one closed std::variant

```
variant<monostate, bool, int64_t, double, user_inline, user_ref>   ·   32 bytes
```

Inline: nothing to manage

`std::monostate`	`bool`
`std::int64_t`	`double`

`user_inline: POD structs ≤ 16 bytes`

e.g. PosPair and Voice in the DAWmulator. monostate means “no value”.

user_ref: pooled cells



one TLSF pool per region, fixed size

- Ref-counted; freed on its home thread once no other thread can still read it
- TLSF: an allocator designed for real-time constraints

Roadmap

No priority order, just open directions.

- **Framework integration**

  e.g. Visage. The original motivation.

- **Sugared API**

  Named bundles over zip and lift.

- **Dynamic graphs**

  Rebuilt and swapped off the audio thread.

- **Error handling**

  A policy for failing ticks.

- **Snapshots**

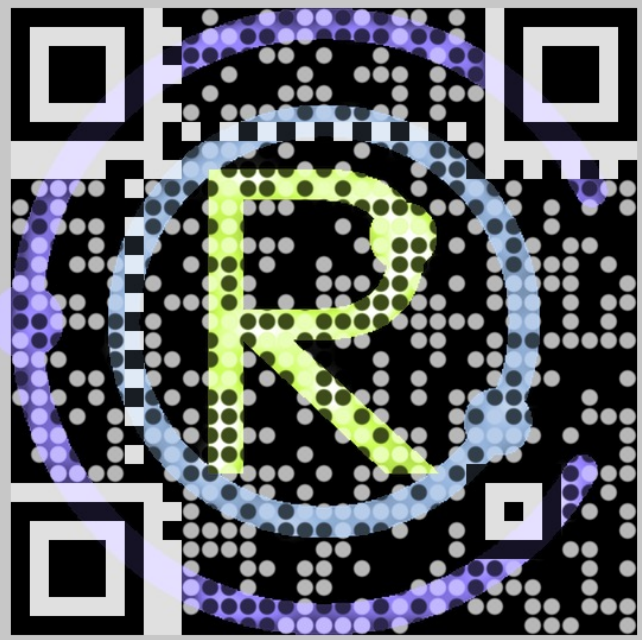
  Undo, replay, time travel.

- **Rough edges**

  Backlog, server, pools, ticks, purity.

THANK YOU!

Questions?



https://clockr.emgware.com