Making Time Uncertainty a First-Class Concept in Linux Timing

Trustworthy timestamps for distributed and AI systems

We make this decision constantly

- event A at T_1 → event B at T_2 → “A happened before B”
  - Ordering. Causality. “Which write won?” “What caused the stall?”
- The hidden assumption: both timestamps are exact points.
  - Clocks disagree. Timestamps are captured at different layers.
  - Synchronization data goes stale between updates.

ROADMAP

Two talks, one story

This talk

- What is the error bar on a timestamp?
- Timestamps as intervals
- Explicit correctness criteria
- “Can I trust this time?”

Prior talk — fast PHC access

- How do we read the clock cheaply?
- PHC reads without syscall / PCIe overhead
- Userspace PHC approximation (Hermóðr)
- ~5 µs syscall+PCIe read → ~60 ns

PART 1

Precise is not the same as correct

We spent two decades chasing precision

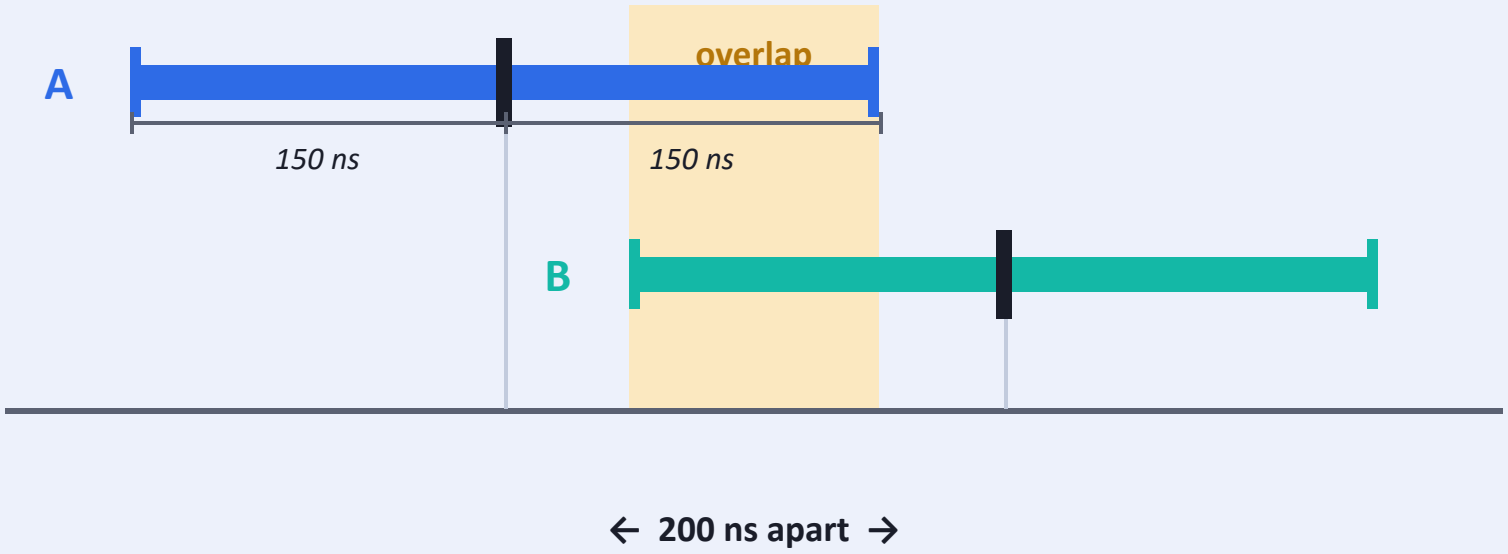
- Sub-microsecond PTP is now common on data-center and AI fabrics
- Hardware timestamping in NICs, SmartNICs, DPUs
- Multi-node GPU traces spanning thousands of ranks
- The better our clocks got, the more we trusted them — implicitly
  - Precision raised confidence faster than it raised correctness

THE PROBLEM

Ways timestamps quietly mislead

Domain	Naive claim	What can go wrong
Ordering	A before B	intervals overlap → order is unknown
Causality	A caused B	off-by-one across nodes → false cause
Compliance	log is ordered	cannot prove ordering is defensible
Consistency	commit A visible before B	overlap breaks external consistency
Profiling	span A nested in span B	cross-node spans misorder
Leases / locks	lease expired before regrant	overlap → split brain / two leaders
Telemetry	counters sampled together	inter-node skew → phantom correlations
Forensics	log A caused log B	causal chain is indefensible
Sensor fusion	radar + camera aligned	physical-world misalignment
Finance	trade timestamped correctly	can't prove mandated window (MiFID II)
Launch time load balancing	packets will have their dedicated slots	packet collisions

Two events 200 ns apart, each known to ±150 ns.



Their true order is unknown — and nothing in the timestamp told you so.

PART 2

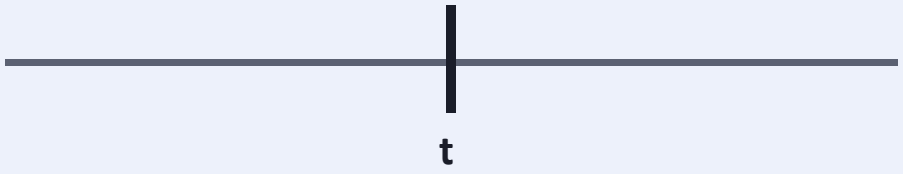
A timestamp is an interval

From a point to an interval

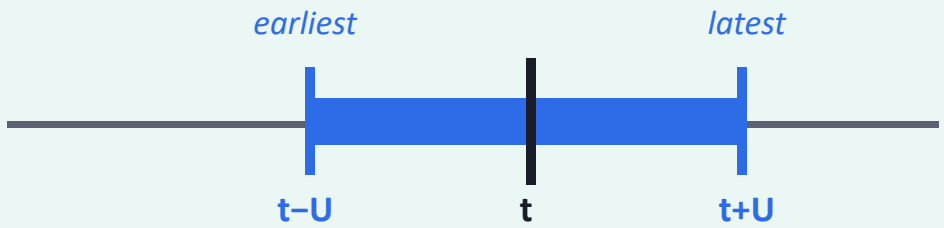
- Instead of: “the event happened at t”
- Say: “the event happened at t ± U”
  - U = the time-uncertainty bound at the moment of observation
  - Also can be viewed as earliest-latest
- Now applications can ask explicit questions:
  - Is ordering between A and B definite, or ambiguous?
  - Is this window fresh enough for a compliance decision?

The ordering rule

Point model



Interval model




```
A before B   only if   tA + UA  <  tB – UB
```

otherwise → ambiguous · unknown, not wrong

The overlap region is where causality cannot be resolved from timestamps alone.

What uncertainty is — and is not

U IS

- A computed uncertainty envelope, right now
- Derived from sync state + explicitly stated assumptions
- An explicit input to application logic

U IS NOT

- A statistical confidence interval (unless you add stats)
- A proven upper bound — some terms are omitted
- Zero just because PTP reports “SLAVE”

PART 3

Where does uncertainty come from?

Walk the stack, bottom to top

```
6  Userspace observation delay   <- syscall + scheduling
5  Kernel -> userspace transfer  <- IRQ, softirq, copy
4  Timestamp capture point       <- HW vs SW
3  Network                       <- queueing, routing, hop count
2  PHC clock                     <- offset, clock resolution
1  Oscillator                    <- drift, temperature
```

Each layer adds error or delay. Only some layers expose metrics.

ANATOMY

The uncertainty budget: every source

Source of error	What it contributes	Where it lands
Grandmaster class	advertised GM quality (clockClass / accuracy)	assumed bound
Offset from grandmaster	estimated bias, from the t1–t4 exchange	U_clock: |offset_est| + U_est
Residual oscillator drift	freq error left after discipline (≤ D_max)	U_clock: age × D_max
Holdover	no discipline (ptp4l down) — age keeps growing	U_clock: anchor held
Clock resolution	granularity of the timestamp tick	U_clock: ½ tick
Path-element delays	estimated mean is compensated by PTP	residual est. error → U_est
Link asymmetry	up and down paths differ	unmodeled → folds into U_est
Event capture point	how the app stamped the event (HW/SW)	U_capture (application)

Layers 1–2: oscillator & PHC clock

- Oscillator (layer 1): TCXO/OCXO stability (ppb), temperature sensitivity
- PHC clock (layer 2): the disciplined hardware clock we actually read
  - offsetFromMaster — an estimate of bias, with its own uncertainty (U_est)
  - clock resolution — ½-tick quantization (effective, not just clock_getres)
  - PHC read latency — the cost of reading it

Layer 4: capture point — a separate uncertainty (U_capture)

Where stamped	Capture uncertainty	Dominant term
PHC hardware RX timestamp	smallest	HW stamp at the wire
SO_TIMESTAMPING (software)	larger	IRQ + softirq delay
clock_gettime() in app	largest	event→read gap (not clock error)

Layers 3 & 5: network and kernel path

- Network (layer 3): each hop adds queueing + routing delay
  - PTP compensates the *estimated* mean path delay — its estimation error remains
  - Residual = asymmetry + delay-estimation error, growing with hops — folds into U_est
- Kernel path (layer 5): Linux exposes useful data — but not as one package:
  - PTP_SYS_OFFSET / _EXTENDED — PHC vs system clock
  - SO_TIMESTAMPING — ingress / egress timestamps
  - ptp4l management API — offset, delay, ingress time (userspace)

Layer that everyone forgets: staleness

```
Between syncs, we bound the residual frequency error.

U_drift  =  age  x  D_max      (configured bound)

Example:  100 ppm  x  1 s   =   100 microseconds
```

Staleness turns a point estimate into a growing envelope — age × the configured frequency-error bound.

PART 4

Turning PTP state into a bound

PTP answers: what does the protocol believe?

Dataset (via ptp4l)	Provides
CURRENT_DATA_SET	offsetFromMaster, meanPathDelay, stepsRemoved
PORT_DATA_SET	port state, logSyncInterval
TIME_STATUS_NP	sync ingress time, grandmaster identity
PORT_HWCLOCK_NP	PHC index (optional)

Ingress time is the drift anchor

```
TIME_STATUS_NP.ingress_time
   = PHC time of the last sync event

drift = (time_now - ingress_time) * D_max / 1e9
      = attributed residual freq-error bound

Caveat: offset and ingress come from *different*
datasets - no common observation epoch.
```

The model — stated with its assumptions

```
U_event  =  U_clock            +  U_capture
            (daemon / library)     (application)

U_clock  =  |offset_est|          # estimated bias, uncorrected
         +  U_est                 # uncertainty of that estimate
         +  age x D_max           # attributed drift since anchor
         +  res / 2               # clock / timestamp quantization

Assumptions:  |freq err| <= D_max ,  |asymmetry| <= A_max ,
              |offset-est err| <= U_est
```

An operational envelope under stated assumptions — not a proven upper bound on true clock error.

PART 5

A model implementation: ptp-uncertainty

Architecture

```
ptp4l
  |  Unix management socket (poll offset/delay/ingress)
  v
ptp_unc_dmn        <- daemon: collects + anchors
  |  /ptp_uncertainty (POSIX shared memory)
  v
libptp_unc.so      <- client lib: extrapolates at read time
  |
  v
applications       <- get a live bound, no PTP code needed
```

No direct PHC access required. A live bound, not just the last offset snapshot.

What the daemon collects

Field	Source
offset, delay, steps removed	CURRENT_DATA_SET
port state, sync interval	PORT_DATA_SET
ingress time, grandmaster id	TIME_STATUS_NP
PHC index	PORT_HWCLOCK_NP

Client API

```
struct ptp_unc_handle *h = ptp_unc_open();
struct ptp_uncertainty u;

ptp_unc_get(h, &u);            // U_clock at now

// u.total_uncertainty_ns  = U_clock (daemon)
// u.is_synchronized, u.ptp4l_connected
// u.age_ns  (snapshot freshness != drift anchor)

U_event = u.total_uncertainty_ns + U_capture;
ptp_unc_close(h);
```

The daemon returns U_clock; the application adds its own U_capture to get U_event — the daemon cannot know how you stamped.

Behavior under failure = explicit correctness

```
// ptp4l restarts or disconnects:
//   - preserve last valid sync anchor
//   - set ptp4l_connected = 0
//   - drift KEEPS growing from the held anchor

if (!u.ptp4l_connected ||
    u.total_uncertainty_ns > threshold)
    mark_event_ordering_untrusted();
```

The application decides what 'trustworthy enough' means — the daemon just keeps the bound honest.

You don’t have to take my word for it: Meta’s fbclock

Meta fbclock (deployed at scale, 2022)

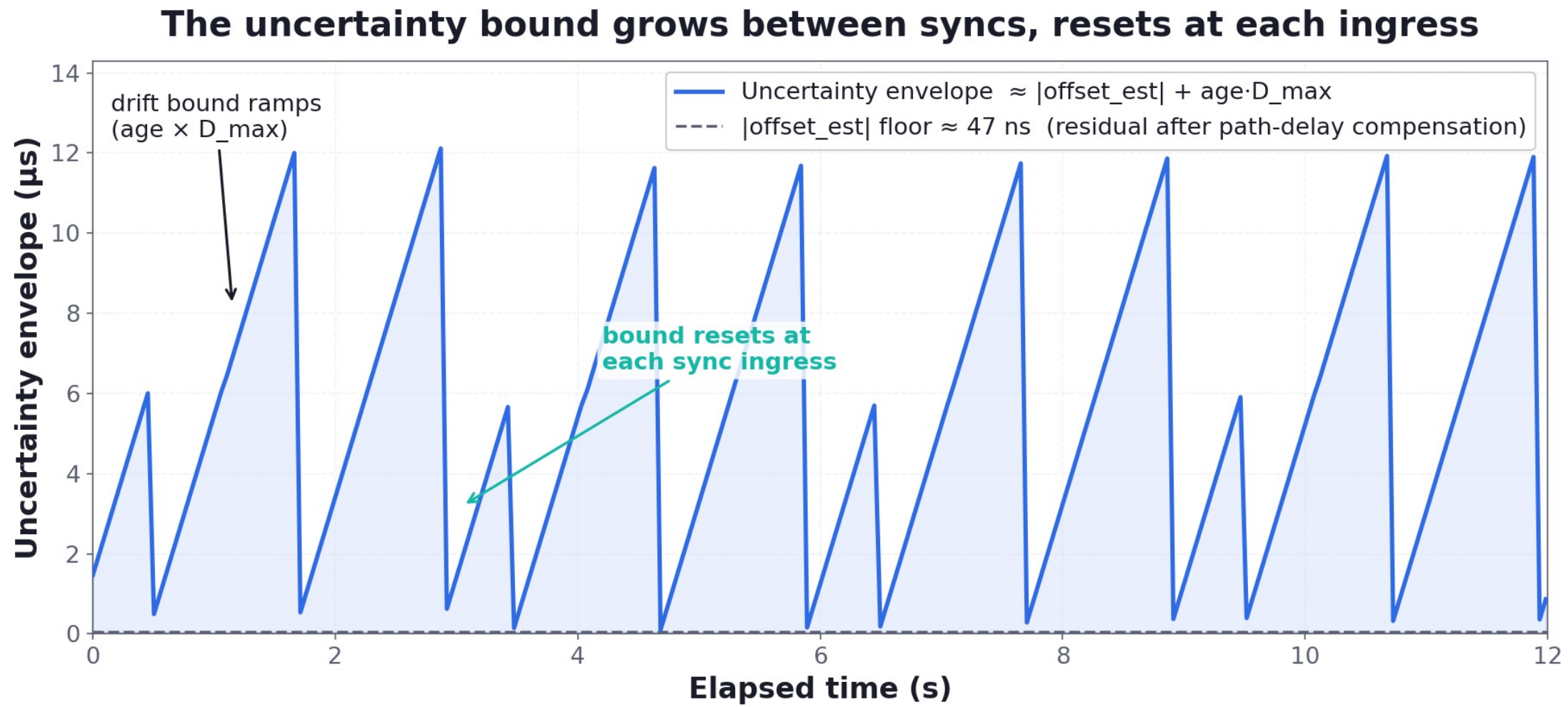
- Returns a Window of Uncertainty: [earliest_ns, latest_ns]
- Statistical k·σ window (Meta targets 6-nines certainty)
- Holdover: drift from PHC freq-adjust history

ptp-uncertainty (this talk)

- Returns total_uncertainty_ns → [t−U, t+U]
- Conservative worst-case bound (layer stats on top if wanted)
- Holdover: hold last anchor, drift grows at max_drift_ppb

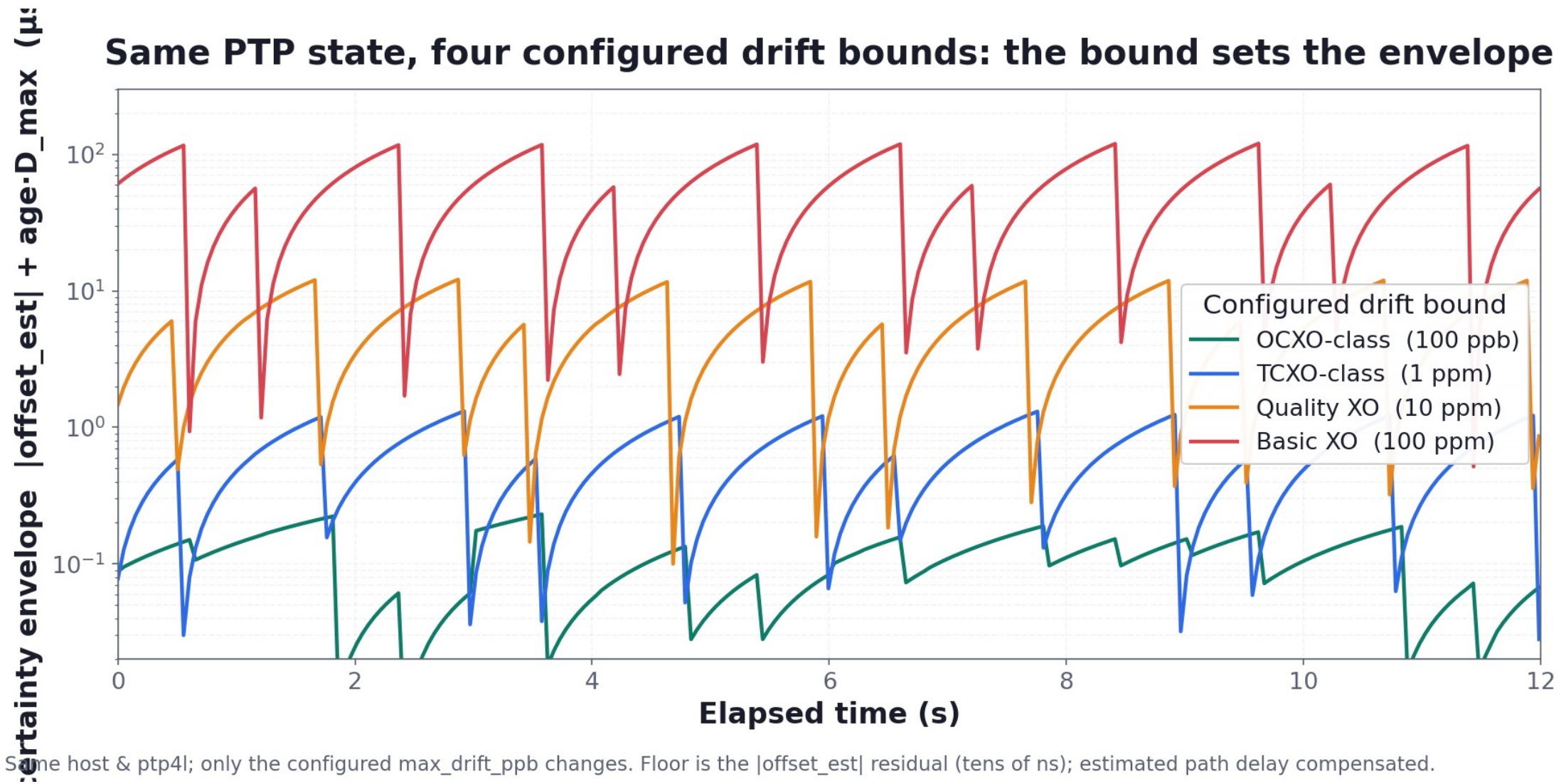
PART 6

Model behavior across oscillator bounds



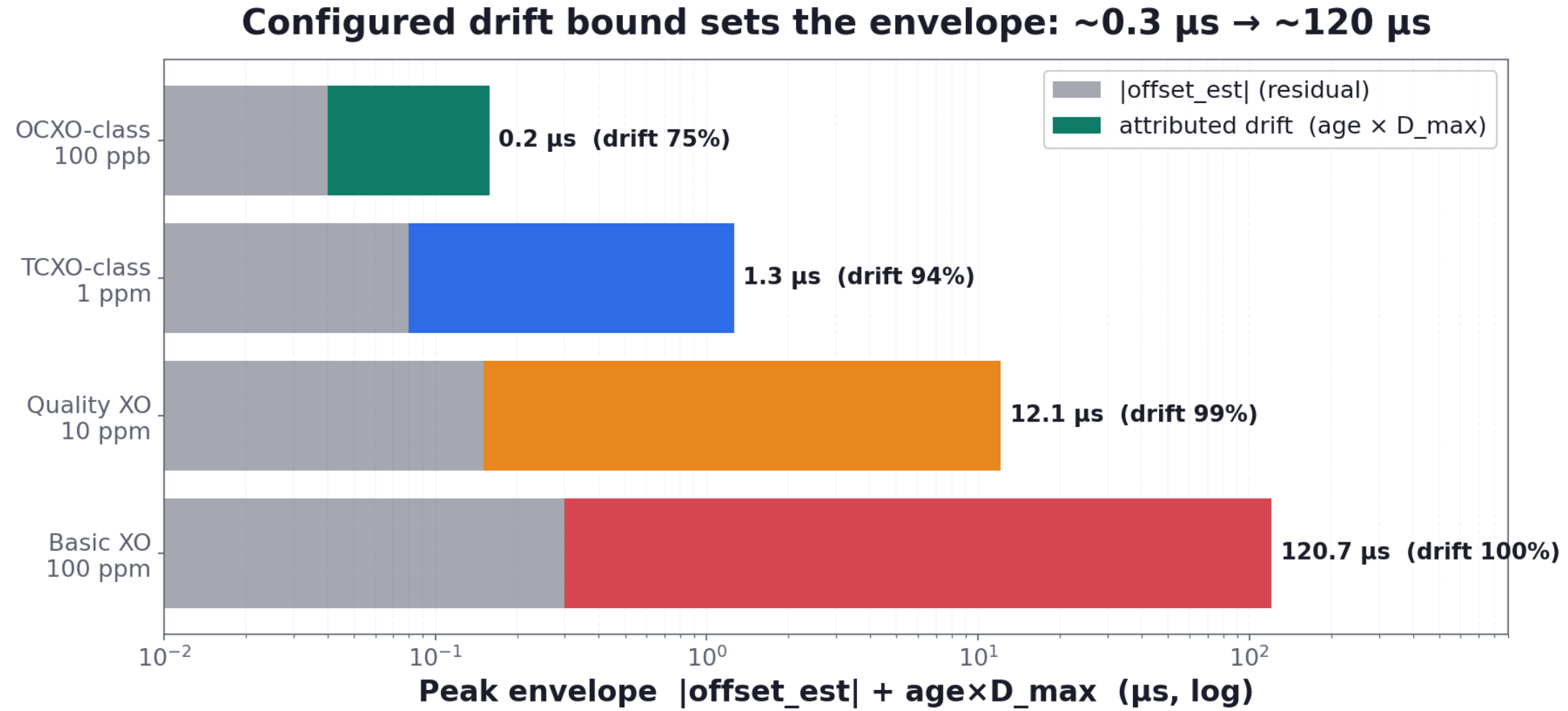
Configured drift bound D_max = 10 ppm · same host, ptp4l @ 1 Hz · estimated path delay compensated · watch_uncertainty -raw

Between syncs the bound grows as age × D_max, then resets at ingress — this is the model's attributed drift, not measured drift.



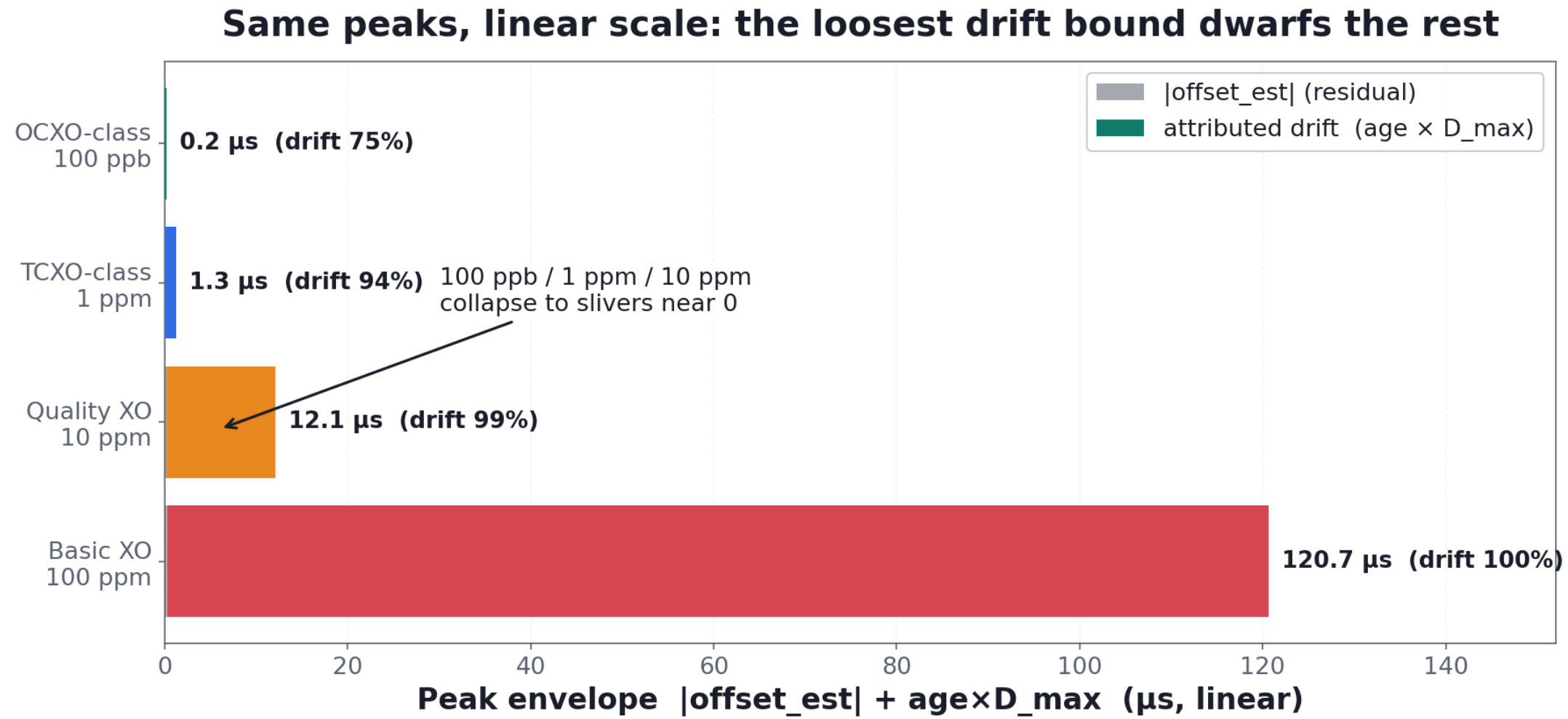
Same host & ptp4l; only the configured max_drift_ppb changes. Floor is the |offset_est| residual (tens of ns); estimated path delay compensated.

Same PTP state, four configured drift bounds — the assumed D_max alone spans two orders of magnitude.



Worst-case moment · same host, four configured D_max · estimated path delay compensated · envelope grows with the configured drift bound.

Between syncs the configured drift bound — not PTP — sets whether the envelope is 0.3 µs or 120 µs.



Linear x-axis. Same host & ptp4l, four configured D_max — 100 ppm (~120 µs) makes the tighter bounds look flat.

Between syncs the configured drift bound — not PTP — sets whether the envelope is 0.3 µs or 120 µs.

PART 7

The full story: fast time + its uncertainty

Fast access gives you the time cheaply. Uncertainty tells you how far to trust it.

The prior talk made the read fast; this one puts an error bar on it — fast time and its uncertainty, together.

Fast time + its uncertainty compose

Capability	Fast read alone	+ uncertainty
Low-overhead timestamp	yes	yes
Compare two events	point compare	interval compare
Know when order is ambiguous	no	explicit
Gate decisions on trust	no	threshold on U
Merge / correlation window	fixed	adapts to live U

PART 8

Linux APIs: what we have, what we lack

The kernel gives ingredients, not the recipe

Kernel today: raw ingredients

- SO_TIMESTAMPING — HW/SW RX/TX timestamps
- PTP_SYS_OFFSET(_EXTENDED) — PHC ↔ system
- /dev/ptpN — freq adjust, ext timestamps
- sync state lives only in ptp4l userspace

Missing: a coherent clock-state snapshot

- one atomic read at a defined epoch
- master_offset_ns + ingress_time_ns (anchor)
- max_drift_ppb — assumed frequency-error bound
- gm_id + steps_removed (per-hop = policy, not a bound)
- → enough metadata to derive U under stated assumptions

CLOSING

Takeaways

- Timestamps are intervals — design systems around t ± U
- U is an envelope under stated assumptions, not a proven bound
- Separate U_clock (daemon) from U_capture (application)
- Staleness is an attributed drift bound — the term most tools ignore
- Missing kernel primitive: a coherent clock-state at one epoch
- The full story: fast access to the time and its uncertainty

Questions?

Treat time as an interval. Make the error bar part of the data.

`ptp-uncertainty` · `daemon + libptp_unc.so + watch_uncertainty`