PROCESSOR API · PER-KEY TIME

Ask for a timer. Don't build one.

A StoatFlow timer is a one-shot callback for one key at one time. Register it from `process()`; `onTimer()` runs for that key, on that key's lane, in order with its records. Timers are stored, restored after a restart and exactly-once by default. No index store, no scan, no delete loop to write.

1 transaction
Output, delete, offsets
A fire's output and the timer's deletion commit with the input.

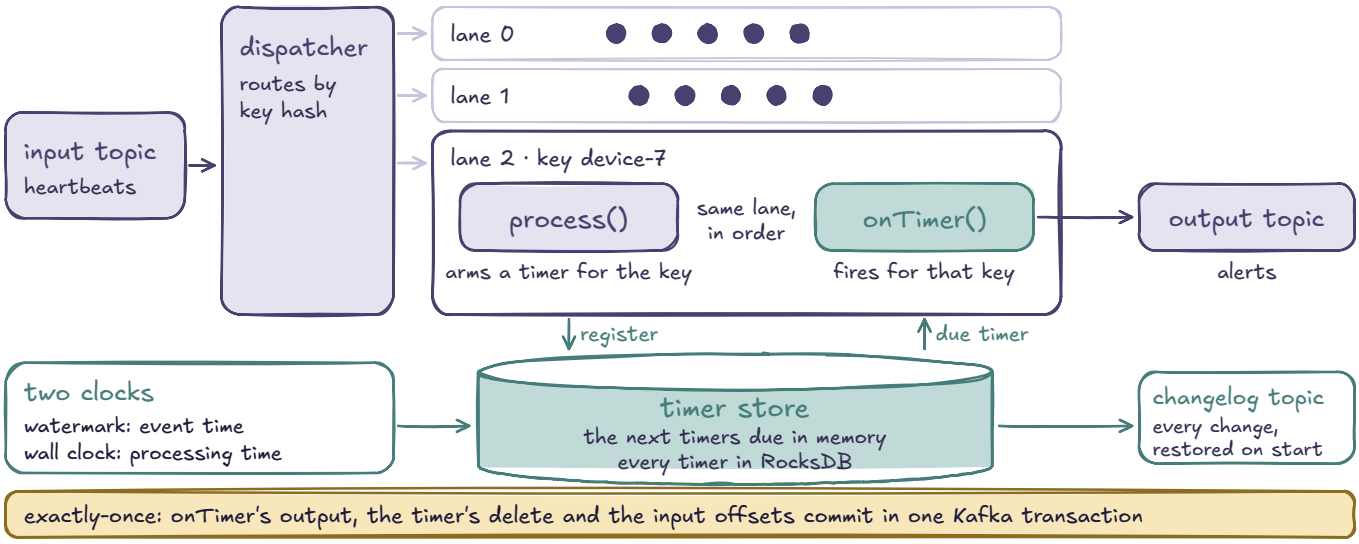
2 clocks
Event and processing time
The watermark or the wall clock, declared per processor.

100K per clock
Timers held in memory
The default bound; the rest wait in RocksDB.

1 call
To move a deadline
Register a named timer again and it moves.

HOW IT WORKS

The timer runs where the key's records run



Records are routed by key hash to lanes, one thread each. When the watermark (event time) or the wall clock (processing time) reaches a timer, it fires on its key's lane, so `onTimer()` and `process()` never race for a key. A timer can be armed for any key; it fires on that key's lane.

EXACTLY ONCE, BY DEFAULT

Everything a timer does commits with the input that caused it

Under `processing-guarantee: EXACTLY_ONCE`, the default, `onTimer()`'s output, its state writes, the timer's own deletion, any timer it registers and the input offsets commit in one Kafka transaction, or not at all. A `read_committed` consumer sees each timer's effect once, even across crashes: if the process dies before the commit, the timer is still stored and fires again after the restart.

Not covered: side effects outside Kafka. `onTimer()` can run again after a crash, so make external calls idempotent, as in `process()`. Under `AT_LEAST_ONCE` a timer can repeat its output but never loses it. The `HEAP` backend keeps no timers across a restart.

TWO KINDS OF TIMER

Regular timers fire at a time; named timers can move

	REGULAR	NAMED
Identity	key, timestamp and clock	key and name, across both clocks
Register again	same key and time: still one timer	replaces it: a new time, or the other clock
Delete	`deleteEventTimeTimer(key, ts)`	`deleteTimer(key, name)`, `deleteAllNamedTimers(key)`
Look up	by time range via IQv2, planned for 1.1.0	`namedTimer(key, name)`: the clock and time, or `null`
Good for	one action at a fixed time	deadlines that move: timeouts, debounce, retries

> The two are separate: a named and a regular timer for the same key and time both fire. Both carry headers: the arming record's, plus any you pass, come back in `ctx.headers()`, and `ctx.forward(key, value)` carries them, so a trace id survives the wait.

THE API

A `TimerService` from `context.timerService()`, an `onTimer()` on the processor

`register{Event|Processing}TimeTimer(key, ts)`	One-shot. It fires once its clock reaches `ts`; at or before now, at the next pass.
`register…Timer(key, name, ts)`	Named: replaces the key's timer of that name, in either clock.
`register…Timer(key, [name,] ts, headers)`	Your headers over the arming record's, one value per header key.
`delete…Timer(key, ts)` · `deleteTimer(key, name)`	Exact delete, or by name; `deleteAllNamedTimers(key)` clears a key's named timers.
`namedTimer(key, name)`	The pending named timer's clock and time, or `null`.
`declaredTimerTypes()`	On the processor: the clocks it uses. Registering on another throws.
`onTimer(ts, key, ctx)`	`forward(…)`, `headers()`, `timerName()`, `timeDomain()`, `getStateStore(…)`

TWO EXAMPLES, IN JAVA

Heartbeat timeout: a named processing-time timer

```
private ProcessorContext<String, String> context;
@Override public Set<TimeNotion> declaredTimerTypes() {
    return Set.of(TimeNotion.PROCESSING_TIME);
}
@Override public void init(
        ProcessorContext<String, String> context) {
    this.context = context;
}
@Override public void process(Record<String, String> r) {
    TimerService<String> timers = context.timerService();
    timers.registerProcessingTimeTimer(r.key(), "silence",
        timers.currentProcessingTime() + 60_000);
}   // named: the next heartbeat moves the deadline
@Override public void onTimer(long ts, String device,
        TimerContext<String, String> ctx) {
    ctx.forward(device, "no heartbeat");  // at time ts
}
```

```
// wired from the DSL
builder.stream("heartbeats",
        Consumed.with(Serdes.String(), Serdes.String()))
    .process(HeartbeatTimeout::new)
    .to("alerts",
        Produced.with(Serdes.String(), Serdes.String()));
```

Delayed retry: the attempt count in the timer's headers

```
@Override public void process(Record<String, String> r) {
    if (send.test(r.value())) return;
    boolean waiting = pending.get(r.key()) != null;
    pending.put(r.key(), r.value());  // the latest wins
    if (!waiting) arm(context.timerService(), r.key(), 1);
}
@Override public void onTimer(long ts, String key,
        TimerContext<String, String> ctx) {
    int attempt = attemptOf(ctx.headers());
    String req = pending.get(key);
    if (send.test(req)) { pending.delete(key); return; }
    if (attempt < MAX_ATTEMPTS) {
        arm(ctx.timerService(), key, attempt + 1); return;
    }
    ctx.forward(key, req);   // dead letter, trace id kept
    pending.delete(key);
}
static void arm(TimerService<String> t, String k, int n) {
    t.registerProcessingTimeTimer(k, "retry",
        t.currentProcessingTime() + (BASE_MS << n),
        new RecordHeaders().add("x-attempt", bytes(n)));
}   // backoff 2 s, 4 s, 8 s ...; the attempt rides along
```

Timers live in the Processor API: `process(…)` or `processValues(…)` from any stream. With no input at all, `builder.scheduled(…)` fires on a schedule, not per key. In tests the `TopologyTestDriver` drives both clocks: `advanceWatermark(ts)` and `advanceWallClockTime(d)`.

CONFIGURE

```
stoatflow:
  processing-guarantee: EXACTLY_ONCE  # the default
  timer-backend:
    type: HYBRID               # or PERSISTENT, HEAP
    max-cached-timers: 100000  # per clock, in memory
    cache-horizon-ms: 60000    # HYBRID: load 60 s ahead
    changelog-enabled: true    # restore from Kafka
```

```
.timerBackend(new TimerBackend.TwoTier(
    Duration.ofSeconds(60), 100_000))   // wins over YAML
```

BACKEND	WHERE TIMERS LIVE	RESTART
`HYBRID` (default)	RocksDB; the next 60 s in memory, at most `max-cached-timers` per clock	Restored
`PERSISTENT`	RocksDB; the next second in memory, same bound	Restored
`HEAP`	Memory only, unbounded: tests, or timers you can rebuild	Lost

One backend per application. While more event-time timers are due than fit, event time waits at the first timer not yet loaded, so none fires late.

INSPECTING TIMERS

- `namedTimer(key, name)`, inside a processor: is one pending, and when.
- `GET /watermarks`: `timerFrontierMs`, `timerHeld`, `timerHeldSinceMs`.
- `GET /topology/compiled`: which processors register timers.
- There is no interactive query over pending timers yet.

METRICS

- `stoatflow.timer.registered|fired|deleted.total`, by clock
- `stoatflow.timer.cache.loaded`, `…cache.overshoot.max`: memory
- `stoatflow.watermark.timer.held`, `…frontier`: event time held
- `stoatflow.timer.lane.mismatch.total`: keys on another lane

Try it on your own deadlines

Every pattern in the talk ran on StoatFlow. For task-wide periodic work Kafka Streams' punctuators are simpler, and StoatFlow keeps them as they are. Free 30-day trial.

Docs	**stoatflow.io/docs/building/processor-api**
Book a call	**cal.com/hartmut-stoatflow-io/get-in-touch**
Email	**sales@stoatflow.io**