VELOCITY EDGE

TICK-TO-TRADE, PART 1

Three Questions to Ask of Any Latency Figure

ABSTRACT

Latency figures for trading systems are quoted freely and defended rarely. This paper, the first of a series, puts three questions to any such figure: which two events it runs between and which clocks stamped them; which samples were taken, and when; and how it was summarised. Each is a test, and a figure that fails any of them can mislead, however carefully the rest was done.

We define tick-to-trade so that it can be measured rather than asserted, lay out the milestones and stages of a market-making path, and say what a cloud host, a co-located one and a path built in hardware can each observe of them. We then argue that latency is a distribution, not a number, and that its most valuable property is often how tightly its tail is bounded, which the series calls deterministic latency. The paper also sets out what an honest tick-to-trade measurement requires, why percentiles cannot be averaged, and what coordinated omission hides.

The series is planned in nine parts. Together they will answer each question for Darkwave's Rust engine, built to make markets in crypto perpetual futures from a cloud instance; show how the method carries over to a co-located host and to a path built in hardware; and measure what a microsecond is worth.

Phillip Pienaar
Velocity Edge

2026-10-24
Version 1.0.0

Contents

Executive Summary 3
1. A Latency Figure Is a Claim 4
2. Tick-to-Trade, Defined So It Can Be Measured 5
3. The Wrong Clock 12
4. Latency Is a Distribution, Not a Number 14
5. The Missing Samples 17
6. The Averaged Percentile 19
7. What This Series Measures 21
Conclusion 24
Glossary 24
References 25
Revision History 29
Licence and Disclaimer 30

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Executive Summary

A head of technology sees latency figures all the time: in a vendor's deck, in a design review, on their own team's dashboard. The figure is usually a single number with a percentile attached, such as the p99, the latency that 99 % of observations do not exceed, and it is often accepted without a second look. This paper gives three questions to ask of it (Figure 1); Sections 2, 3, 5 and 6 show what a failing answer to each looks like. The rest of the series will ask the same questions of a real engine, Darkwave's, and publish the answers, withholding its strategy, its earnings and where its host sits, as Section 7 sets out.

- **Which two events does the figure run between, and which clocks stamped them?** A figure that ends at the send call but is quoted as the order leaving the host fails the first half. The series' own figure ends at the order's hand-off to the network interface, and says so. A figure whose two stamps come from clocks that were never compared fails the second half. Clocks on different machines compare only to within their clock error bounds, read and logged with the stamps, and a venue that publishes no bound for its clock gives none.
- **Which samples were taken, and when?** Stamping every message is not enough. Market-data updates stamped when they were read, not when they arrived, hide their wait in the kernel. An engine that falls far enough behind pushes the wait back to the venue, where no stamp sees it. A figure counted only over updates that led to an order drops the ones that came too late, which is why the series also reports tick-to-decision, from each update to the decision that took it into account. And a figure from a quiet hour says nothing about the busiest minute.
- **How was it summarised?** A percentile cannot be rebuilt from other percentiles. Averaging twenty-four hourly p99s does not give the daily p99, which may sit anywhere between the smallest of them and the largest. The honest method keeps the full distribution, merges distributions, and takes the percentile once.

Darkwave's engine is built to make markets in crypto perpetual futures. In 2025 perpetuals traded about $62 trillion of volume on the exchanges CryptoQuant tracks, against $18.6 trillion for spot, more than three times as much [1]. A perpetual contract never expires and trades around the clock, and its venue is reached over a public API from a cloud region, through a verified account rather than an exchange membership and a clearing broker. That openness gives a firm in the public cloud the same access as its peers. Their instances, placement and network paths can still differ, so a firm's own path, and how that path is measured, decide much of the race.

The questions matter because a latency distribution is not fixed. Under bursts of market data the median barely moves, but the rare slow cases, the tail, get slower, so the p99.9 and the maximum grow. We expect bursts when prices move, exactly when quotes most need to change. The first update after a quiet period can also be served more slowly than the ones that follow it. For a market maker on a cloud instance, behind a venue gateway it does not control, how tightly this tail is bounded can therefore matter more than the median. The series calls that property deterministic latency, in the weaker sense of a bounded tail rather than the fixed cycle count the term means in hardware.

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

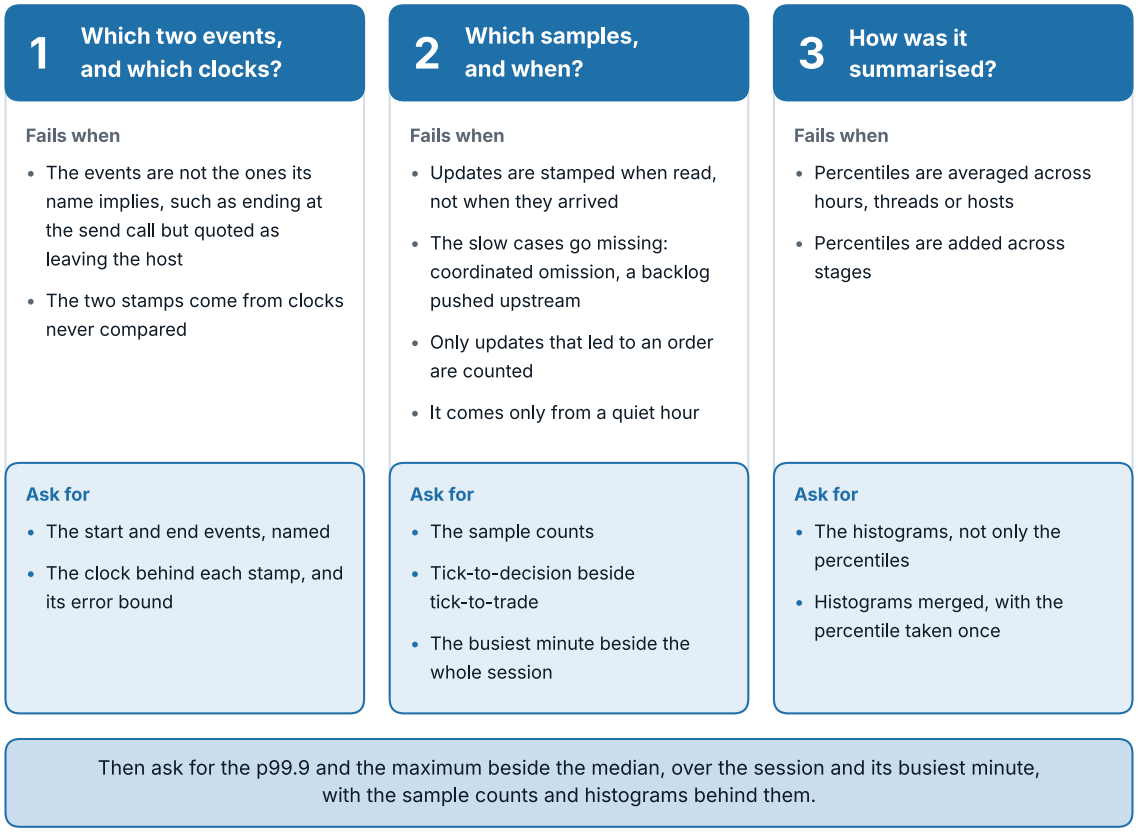


Figure 1: The three questions to ask of any latency figure, what each catches, and what to ask for instead; a schematic. Sections 2 and 3 take up the first (Figures 2 to 4), Section 4 shows why the second and third matter (Figures 5 and 6), and Sections 5 and 6 take up the second and third (Figures 7 and 8). Source: figures/part-1-figures.py.

Of the whole round trip, the tail of the firm's own path is what its engineering can most directly change.

Of any figure, then, ask which two events it runs between and which clocks stamped them. Then ask for the p99.9 and the maximum beside the median, over the whole session and over its busiest minute, with the sample counts and the histograms behind them.

1. A Latency Figure Is a Claim

Every latency figure rests on three premises: that its two events were named and the clocks that stamped them were compared; that the sample it was drawn from was representative; and that the arithmetic used to summarise that sample preserved what mattered. When a premise fails, the figure can be wrong by orders of magnitude, as Gil Tene's worked examples show [2]. Most such failures can mislead in either direction. Coordinated omission, a failure of the second premise described in Section 5, can only flatter, because the samples it loses are the slow ones.

This matters in trading because speed bears on market quality and decides races. In NASDAQ order-level data from October 2007 and June 2008, low-latency activity is associated with tighter bid–ask spreads and deeper books [3]. In a continuous market, a resting quote that has gone stale is a race between the maker who pulls it and

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

the takers who hit it [4]. Aquilina, Budish and O'Neill, studying message data from the London Stock Exchange, find such races lasting 79 µs on average, with the most common lasting 5 to 10 µs, and accounting for about a fifth of trading volume in FTSE 100 stocks [5]. Races that short are decided among systems placed beside the venue's matching engine, whose reaction times differ by microseconds. An engine on a cloud instance, reaching a venue over TLS, competes on a different footing. Its path crosses the provider's network and the venue's public gateway rather than a cross-connect, the cable that joins a co-located firm's equipment to the venue's. Its round trip is therefore longer and more variable than a co-located one's, a point Section 4 returns to.

For one part of this problem the industry has a regulatory bar. The European Union's rules on business clock synchronisation are now Commission Delegated Regulation (EU) 2025/1155, which replaced Delegated Regulation (EU) 2017/574 from 2 March 2026. Under them, a firm using a high-frequency algorithmic trading technique must keep its business clocks within 100 µs of UTC and use a timestamp granularity of 0.1 µs or finer (Annex IV, Table 2), where the earlier rule required 1 µs or finer (its Annex, Table 2). The firm must also review the traceability of its clocks at least once a year (Article 16) [6], [7]. The rule governs timestamps for reporting rather than for latency measurement, and the engine this series studies, the subject engine, is built for a crypto venue outside its reach. The series adopts the rule anyway, as the bar a defensible timestamp should clear.

For the latency of a whole trading system, one industry reference point is a benchmark rather than a regulation. STAC's T0 benchmark measures tick-to-trade network I/O latency: from simulated UDP market data sent to the system under test to the TCP order it returns, both timestamped on the wire by capture equipment outside the system [8]. That is the standard of observation this series aims at, and Section 2 is plain about where a cloud host falls short of it.

None of this is peculiar to market making on an order book. A dealer's last-look hold time, in which a client's request is held and then accepted or rejected, fails the second question when the rejected requests are left out of its sample. A payment authorisation, a pre-trade risk check and a market-data feed handler are all paths with a start, an end and stages between, and their figures fail in the same ways. The series does not cover links between distant sites, over fibre or microwave; its subject is one host against one venue.

The rest of this paper takes the questions in order. Section 2 names the events and Section 3 the clocks, the two halves of the first question. Section 4 shows that latency is a distribution, which is what gives the second and third questions their force. Section 5 answers the second and Section 6 the third. Section 7 sets out what the series publishes and withholds, its units and its parts.

2. Tick-to-Trade, Defined So It Can Be Measured

The phrase tick-to-trade is used loosely. For the figure to be measurable, both ends must be observable events, each with a known clock. This section defines the start,

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

the tick and the end; names the milestones and stages between them; and then sets out what each kind of host can observe.

Start: the update arrives at the instance. Many crypto venues stream market data from `wss://` endpoints [9], [10], [11], that is, WebSocket over TLS over TCP [12, Secs. 1.7, 3, 4.1]. On such a stream one update can span several TCP segments, and one segment can carry several updates, so "the packet carrying the update" is not well defined. A TLS record, however, can be used only whole. An AEAD (authenticated encryption with associated data) decryption returns the plaintext only if the whole record, tag included, authenticates [13, Sec. 2.2], and TLS 1.3 protects each record this way [14, Sec. 5.2], [15, Sec. 5.2]. We define the start, M0, as the arrival of the data that completes the TLS record holding the end of the update (Figure 2a). That record may end after the update does, and its size is the venue's choice, so the definition excludes the time from the update's last byte arriving to the record's, which varies by venue and stream. It also excludes the time a packet waits at the interface, set out below. Both exclusions flatter the figure, and both are stated.

A receive timestamp, or stamp, belongs to a packet, not to an update, and the two do not line up neatly (Figure 2). Here a read means `recvmsg`, the call that returns stamps beside the data; a TLS library that reads the socket itself hides them, so the application owns the receive loop below TLS. When one read returns several packets, Linux reports the stamp of the segment that held the last byte returned [16]. When packets are merged, by generic receive offload (GRO) [17], [18], [19] or in the socket's receive queue [20], they share one stamp. Between them, some updates' ingress, the stage from M0 to the read, comes out shorter than it was and, with a hardware stamp, others' longer. The rule for attributing stamps to updates, and its bias, are part of the figure's definition. A record that holds several updates gives them one M0. Part 3 will set out the rule the subject engine uses and what it costs.

Where that arrival is stamped decides what the figure contains. A kernel software stamp is taken after the network interface has raised its interrupt or been polled, so the time a packet waits at the interface, including any interrupt moderation the driver applies [21], falls before the stamp and outside the figure. That wait should be stated, or minimised and reported. Interrupt affinity, receive offloads and busy polling each change it, and Part 6 will measure what each removes. On EC2, hardware packet timestamping stamps each incoming packet before it reaches the kernel [22]. It needs three things [23]. The instance must belong to a supported family of the seventh generation or later. It must run Amazon's own Elastic Network Adapter (ENA) driver, version 2.10.0 or later, with its PTP (Precision Time Protocol) hardware clock enabled, which is off by default [24]. And, except for a short list of families in a few legacy-access regions, it must be launched in a precision-time placement group. The interface's receive filter must then be set, and the socket must ask for raw hardware stamps [25]. That moves the start as close to the wire as a cloud instance can reach.

What a tick is. Some crypto market-data streams push each change as it happens. Others, including many order-book depth streams, push the accumulated changes at a fixed cadence. Bybit pushes its linear-contract book every 10 ms at depth 1 and

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

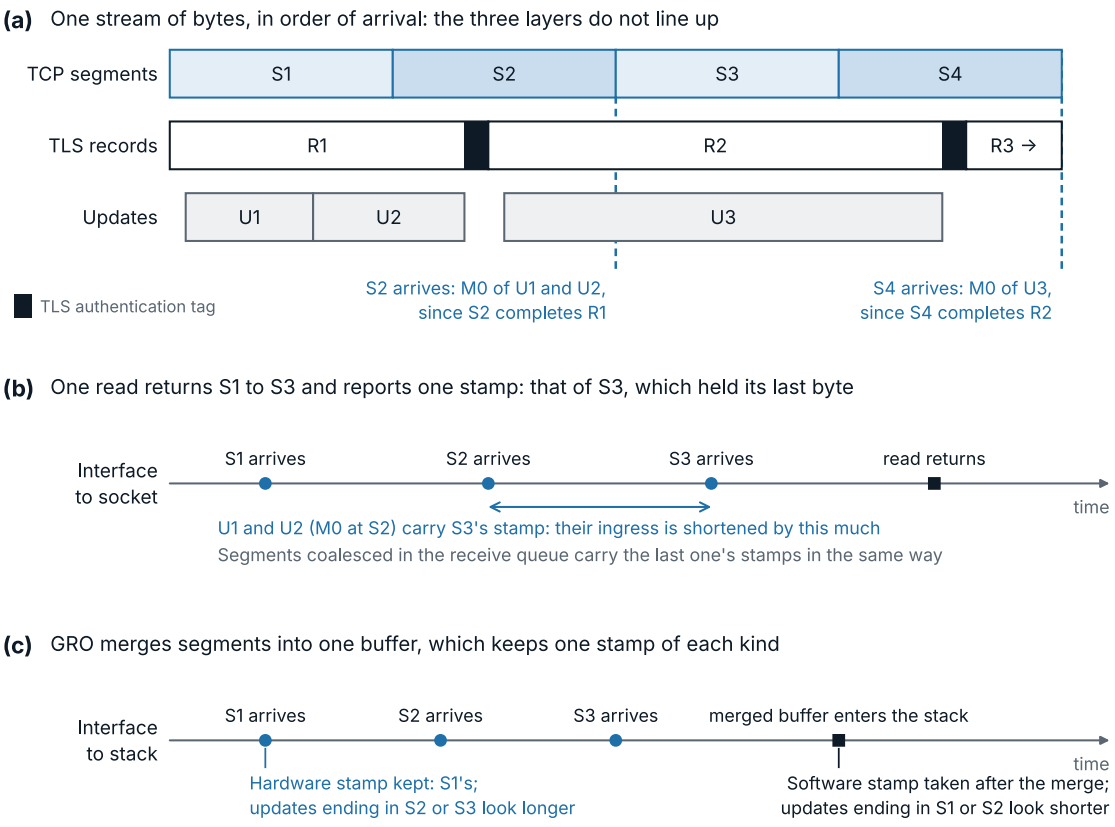


Figure 2: How receive stamps attach to updates; a schematic, not to scale, with illustrative segments, records and updates. (a) One byte stream as TCP segments, TLS records (each ending in its authentication tag) and the updates inside them. An update's M0 is the arrival of the data that completes the TLS record holding its end, so U1 and U2 share S2's arrival and U3 takes S4's. (b) When one read returns several segments, Linux reports the stamp of the segment that held the last byte returned, so earlier updates carry an arrival later than their own [16]; segments coalesced in the socket's receive queue carry the last segment's stamps in the same way [20]. (c) Under generic receive offload the software stamp is taken after the merge, no earlier than the last segment's arrival [18], while the hardware stamp stays on the first segment, because later segments are appended without their stamps [19] (our reading of the kernel code). Source: figures/part-1-figures.py.

every 20 ms at depth 50 [26], and Binance its spot depth streams every 100 ms or 1,000 ms [10]. The tick in tick-to-trade is the venue's message, so any batching the venue does belongs to the venue's latency, not the firm's. A firm racing for stale quotes subscribes to the fastest stream the venue offers. M0 is defined the same way for every stream a figure covers, book depth or trades. Orders go the other way over a persistent WebSocket connection of their own, which the encode stage and the loss recovery below assume.

End: the order is handed to the interface. The end, M5, is the kernel's hand-off of the order's last segment to the instance's network interface. It is the closest observation of the order's departure that a kernel stack on the instance can make, and it comes before departure. The interface's own queueing, including any wait the instance's packet-rate or bandwidth allowance imposes, lies after it and outside the figure.

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Which ticks are counted. Most updates lead a market maker to change nothing, so tick-to-trade can be measured only over updates that produced an order. That sample is conditioned on an outcome. An engine that falls behind and skips or merges updates removes its slowest cases from it. Tick-to-trade is therefore always reported beside tick-to-decision, the time from each update's arrival to the decision that took it into account, together with the share of updates merged into a later decision and the share never taken into any decision, such as those lost with a dropped connection. Without the last, an engine could look fast by quietly falling behind.

Between M0 and M5 a market-making path passes a sequence of milestones. The milestones are instants; the stages are the intervals between them. The tables give them as a cloud instance using the kernel's network stack sees them, and the host paragraphs below say what changes elsewhere.

Milestone	Event
M0	The data completing the TLS record that holds the end of the update arrives, as stamped by the hardware or the kernel; on a feed without TLS, the packet that completes the update
M1	The application has the bytes: the socket read returns
M2	The update is decrypted, decoded and applied to the application's order book; when several updates are applied before one decision, each has its own M2 and they share one M3
M3	The quoting decision is made: new quotes, or the decision not to change them
M4	The application issues the send call carrying the order
M5	The order's last segment is first handed to the network interface, as the kernel's software transmit stamp reports it: a proxy for departure, taken before the interface is told of the packet, not the frame leaving it; where a decision produces several messages, each has its own M5 and tick-to-trade is reported per message, with the first and the last named

Stage	From	To	What it contains
Ingress	M0	M1	The protocol stack after the stamp, the socket queue, wake-up of the receiving thread, the read; with a hardware stamp, also the wait at the interface, including interrupt moderation, the transfer from the interface, interrupt handling and receive offloads
Decode	M1	M2	TLS decryption, WebSocket framing, parsing the venue's message, applying it to the book
Decide	M2	M3	The strategy's computation: fair value, inventory, the quotes to place
Encode	M3	M4	The pre-trade risk checks the firm's rules and its regulator require; serialising the order and, where the venue requires it, signing it; WebSocket framing and TLS encryption
Egress	M4	M5	The send call, the protocol stack, the queueing discipline and the driver

Tick-to-trade is M0 to M5 (Figure 3). The stages differ in length, and each is governed by something different, so one can move while the others hold still. Where the kernel's network stack is used, as it is by the subject engine, the operating system governs

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

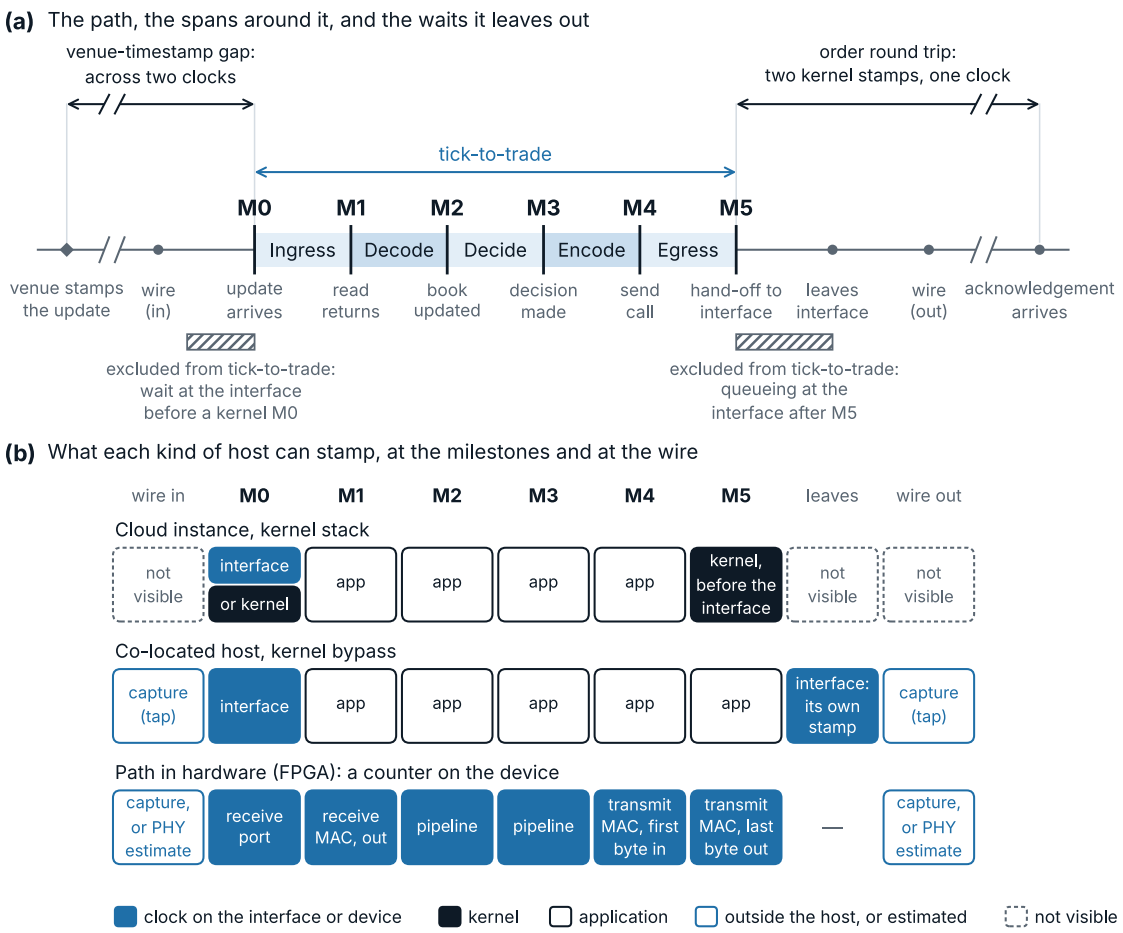


Figure 3: The milestones and stages of tick-to-trade, the spans reported beside it, and what each kind of host can stamp; a schematic, with the stages drawn equal, not to scale. (a) Tick-to-trade runs from M0 to M5. Beside it are the order round trip, from M5 to the arrival of the venue's acknowledgement, both ends taken from the kernel's software stamps on one clock, and the gap between the venue's own timestamp on an update and M0, which spans two clocks; both are drawn broken, without length, because Section 7 withholds their levels. Hatched are the waits excluded from tick-to-trade: the wait at the interface before a kernel M0, which a hardware M0 brings into ingress, and the queueing at the interface after M5. The order round trip includes both: the order's queueing after M5 and the acknowledgement's wait at the interface. (b) On a cloud instance using the kernel's stack, M0 is stamped by the interface's hardware clock where configured, or else by the kernel; the application stamps M1 to M4; M5 is the kernel's software transmit stamp, taken before the interface is told of the packet, a proxy for departure; and the provider's wire is not visible. On a co-located host using a kernel-bypass stack, the interface's clock stamps M0 and the application M1 to M5; the interface's own transmit stamp comes after M5; and capture equipment on a tap stamps both ends on the wire. On a path in hardware, a counter on the device stamps every milestone: M0 at the receive port, M1, M4 and M5 at its media access controllers (MACs), and the stages between in its pipeline; the wire is estimated at its physical-layer interface or captured independently. Source: figures/part-1-figures.py.

ingress and egress; the venue's protocols and the application's code govern decode and encode; and the strategy governs decide. A figure for the whole, without its stages, cannot say which part moved.

Socket options matter as much as code. Nagle's algorithm holds back data that would not fill a full-sized segment while earlier data is unacknowledged [27], [28], and

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

receivers may delay their acknowledgements by up to half a second [28]. Between them they can hold an order back for tens of milliseconds. `TCP_NODELAY` disables Nagle's algorithm [29], but not Linux's automatic corking, which can hold a small write until an earlier packet has left the host and its buffer has been freed on transmit completion. Corking is set by `tcp_autocorking` [30], for the whole network namespace, and the kernel applies it regardless of `TCP_NODELAY` [16]. A figure states both settings.

Loss complicates the end of the path. A lost order segment is recovered by the sender's loss recovery [31], but fast recovery needs later segments to reveal the loss, and an order with nothing sent after it has none. It waits instead for a tail loss probe, or for the retransmission timeout, which doubles on each repeat [32]. On Linux, an order that was the only unacknowledged segment is usually resent about 200 ms plus one round trip after it was sent, unless the minimum timeout is lowered [33]. M5 is the order's first hand-off, so that retransmission lies outside the figure. Loss can also delay an order that was not itself lost. The venue cannot read a new order until an earlier lost segment has been retransmitted, which shows only in the order round trip defined below, and loss recovery can hold the new order in the send queue, which lengthens egress. A tick-to-trade figure is therefore reported with the connection's retransmission count beside it.

What a cloud instance can observe. The instance sees a virtual network interface. The physical wire belongs to the provider, so neither end of the path can be observed on the wire the way the T0 benchmark observes it. Arrival can be stamped in hardware on instances configured for it, or by the kernel. Departure, through the kernel network stack the subject engine uses, can be stamped only by the kernel, and a software transmit stamp is taken "as close as possible, but always prior to" the packet passing to the network interface [25]. The ENA driver takes it after writing the packet's descriptors and before ringing the doorbell that hands them to the device, and when the stack passes packets in a batch, the doorbell waits for the last of them [34]. The stamp is not free: the kernel copies the packet onto the socket's error queue before that doorbell is rung, so asking for it adds to the egress it measures (our reading of the code); Part 3 measures by how much. Orders that share a TLS record or a segment share its M5, as updates that share a record share their M0. The kernel returns the stamp later, on the socket's error queue, and the application must match it to the order, which Part 3 will cover. A figure from a cloud instance therefore states which stamp it used at each end. It is not comparable with a figure taken from the wire, and the series never presents it as one.

What a co-located host can observe. A host in a co-location site beside the venue has physical network interfaces and its own side of the cross-connect. Both ends can therefore be observed close to the wire, in two ways. The interface's own clock stamps packets on receive and on transmit [25], and capture equipment outside the system, fed by a tap on the link, stamps each packet on the wire, as the T0 benchmark does [8]. An interface transmit stamp marks the order leaving the interface, later than M5; it and any stamp from capture equipment are reported as stamps of their own beside M0 and M5, never as either. Where the venue's feed carries no TLS, as multicast feeds

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

do not, M0 is the arrival of the packet that completes the update, and decode does no TLS work. Such a host commonly bypasses the kernel's network stack. The application then stamps M1 itself, when its user-space stack hands it the bytes, and M5, when it hands the order to the interface, by ringing the doorbell for the order's descriptor or, on interfaces that accept it, by writing the frame into the interface's memory. Ingress then holds the transfer into the ring, the poll and the user-space protocol stack, and egress the user-space stack and the descriptor or frame write.

What a path in hardware can observe. On an FPGA the path from wire to wire is logic, not software. There is no kernel and no socket, and a counter on the device stamps every milestone. M0 is the arrival at the device's receive port of the byte the figure starts from, and M1 is that byte leaving the receive media access controller (MAC) for the parser. M2 and M3 are the pipeline's boundaries between decode, decide and encode. M4 is the order's first byte offered to the transmit MAC, and M5 its last byte leaving it. A cut-through design commits an order from fields parsed before the frame has ended, so M4 can precede M1 and the span between them come out negative. A figure states which byte of the update it starts from, and reports such a span as negative rather than as zero. The figure usually quoted is wire to wire, which the device can estimate from stamps at its own physical-layer interface (PHY), but which only capture equipment outside the device can take independently, as the T0 benchmark does [8]. The three questions apply unchanged, with the device's clock and sample in place of the host's. Part 9 takes this up.

Four further measures belong beside tick-to-trade and tick-to-decision.

Measure	Why it matters
Tick-to-cancel	From the M0 of the update that moved the book against a resting quote to the M5 of the cancel: how quickly a stale quote can be pulled before it is hit [4]. Like tick-to-trade, it counts only updates that led to a cancel, so it is read beside tick-to-decision
Loop health	Whether the receiving thread was ready when the update arrived, or still busy with earlier ones, judged by comparing M0 with the instant the thread finished handling the previous update, across the clocks Section 3 names. Updates that share an M0 are judged once, as the first of them. It is an early warning of saturation
Order round trip	From the order's hand-off at M5 to the arrival of the venue's acknowledgement. The series takes both ends from the kernel's software stamps, even where M0 is stamped in hardware, because departure has no hardware stamp. The figure therefore sits on one clock on one instance, subject to the step and slew handling of Section 3. The acknowledgement's software stamp follows its wait at the interface, so the round trip includes that wait. It is the most complete end-to-end figure a cloud instance can observe
Availability	The share of the session during which the path was live; a latency figure covers only the time the path was up

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

3. The Wrong Clock

A latency figure is a difference between two stamps. Section 2 named the events; this section takes the rest of the first question, which clocks produced the stamps.

A Linux system offers several clocks [35]. `CLOCK_REALTIME` tracks wall time. The time daemon adjusts it, by slewing (a gradual change of rate) and by stepping (a jump), so an interval read from it can come out shorter by the size of a backward step, even negative, or longer by the size of a forward one. `CLOCK_MONOTONIC` cannot be stepped, but its frequency is adjusted to track real time. `CLOCK_MONOTONIC_RAW` is not frequency-adjusted, so intervals read from it are untouched by the daemon. Beneath them on an x86 processor is the time-stamp counter (TSC), which on processors with the invariant TSC runs at a constant rate in all power states; Intel's manual adds that the operating system may then use it for wall-clock timer services [36, Sec. 18.17.1]. The subject engine runs on x86 instances, and this part is written for them; an Arm instance has a generic timer in the same role. On a virtual machine the kernel's clock may instead come from a paravirtual clock that the hypervisor maintains for the guest, such as kvmclock [37].

An interval read from one unadjusted clock on one machine, such as `CLOCK_MONOTONIC_RAW` or an invariant TSC, needs no reference to true time. It carries only the clock's own frequency error, which the daemon measures but does not correct on that clock. At tens of parts per million, that is a nanosecond or two over 50 µs. Two conditions apply, which Part 2 will set out: the TSC must be synchronised across the cores the thread may run on, and its reads must be ordered with the work they measure. `CLOCK_MONOTONIC` is slewed like the wall clock, so an interval read from it carries the slew error described below.

But the milestones of Section 2 are not all stamped by the same clock (Figure 4). The kernel's software stamps, for received and transmitted packets alike, are in the system's wall-clock time, which is not necessarily monotonic [25], [38]. Hardware stamps come from the network interface's own clock [25], which on EC2 is the Amazon Time Sync Service's PTP hardware clock [22]. The application reads whichever clock it chooses. So tick-to-trade and ingress span clocks whenever M0 is stamped in hardware, and ingress and egress span clocks whenever the application stamps M1 and M4 from a clock other than the kernel's. The second is the application's choice. Reading the kernel's wall clock at those milestones keeps egress, and ingress with a software M0, on one clock, at the price of that clock's steps and slew.

Whichever choice is made, a defensible figure accounts for the steps and slew the daemon applies to the kernel's clock. It confines steps to start-up, records every adjustment, and discards any interval that spans a step. It states how the kernel's stamps are converted to the application's clock and what error the conversion adds, and how far slewing can move an interval of the length being measured. By default chrony caps the slew rate at one twelfth, 83,333 parts per million [39], which would move a 50 µs interval by about 4 µs. A well-synchronised clock slews away its offset at a few tens of parts per million or less (our estimate; Part 2 measures it). But once steps are confined to start-up, an offset that arises later, however large, is slewed rather than stepped,

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

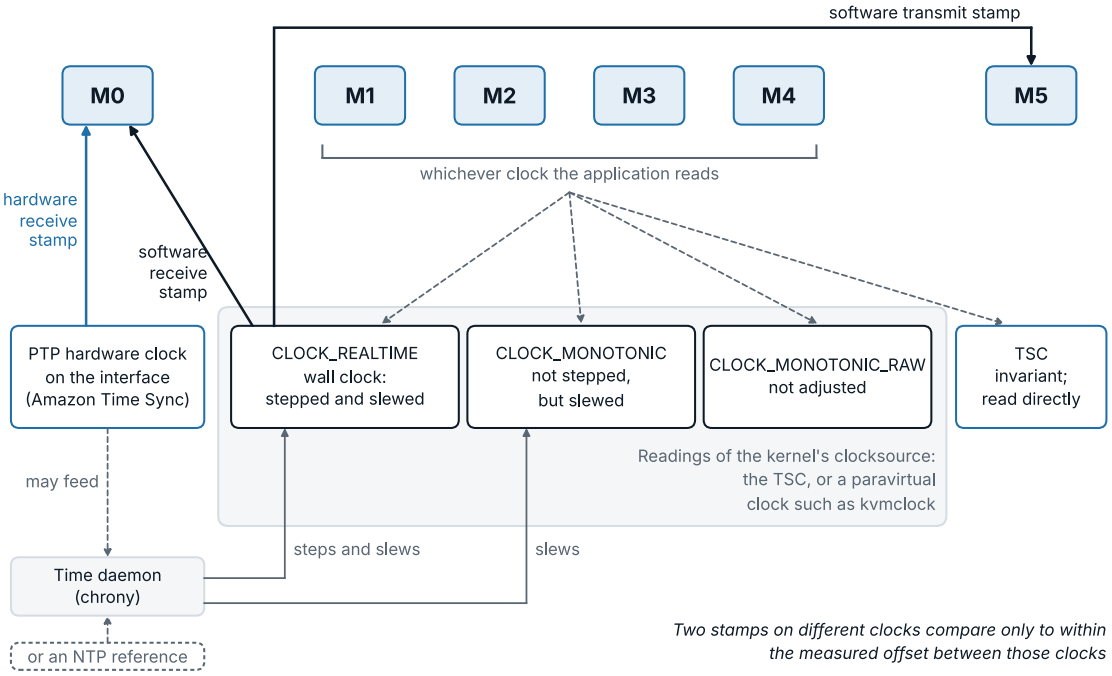


Figure 4: The clocks on one x86 cloud instance, and which of them can stamp each milestone; a schematic. The kernel's software stamps at M0 and M5 are in wall-clock time, CLOCK_REALTIME. A hardware stamp at M0 comes from the PTP hardware clock on the network interface, which on EC2 is the Amazon Time Sync Service's. At M1 to M4 the application reads whichever clock it chooses. CLOCK_REALTIME, CLOCK_MONOTONIC and CLOCK_MONOTONIC_RAW are readings of the kernel's clocksource, which may be the TSC or a paravirtual clock such as kvmclock. The time daemon steps and slews the first and slews the second, and may itself be fed from the hardware clock. Two stamps from different clocks compare only to within the measured offset between them. Source: figures/part-1-figures.py.

at up to the cap. The interval's error is therefore bounded from the rate in force in the daemon's log, not from the typical rate or the cap. Where a stage spans two clocks, as ingress does with a hardware M0, the larger error is usually not slew over the interval but the offset the two clocks have drifted apart since they were last compared.

For a stamp compared with anything off the instance, including the venue's own time-stamps, the clock's relationship to UTC matters too, and it must be known at the moment of the read rather than assumed from the configuration. What can be known of it is the clock error bound: the time daemon's estimate of how far the local clock may be from true time, which changes continuously. Every EC2 instance can reach Amazon's time service over NTP. Supported instances can also reach a higher-accuracy reference over NTP and, with Amazon's ENA driver, a microsecond-accurate one through a hardware clock the Nitro system exposes. Outside a few regions and instance families with legacy access, an instance gets either only if it is launched in a precision-time placement group [23]. AWS reports a clock error bound typically under 100 µs over the higher-accuracy NTP reference and under 40 µs through the hardware clock [40]. Against a bar of 100 µs the NTP figure leaves no margin, since a bound that is typically under 100 µs may at times exceed it, and the hardware clock is what makes clearing the bar credible. Either way, a typical value is not a bound. An instance meets the bar only if the bound is read and logged with every stamp that must answer to it.

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Amazon's ClockBound daemon and library exist for this. They return the current time as an interval, earliest to latest, within which true time lies, with the daemon's clock status [41]. Two events on different machines can be treated as ordered when their intervals do not overlap; otherwise their order is unknown. ClockBound checks a stamp against the local clock; comparing two machines' intervals is the application's work. One trap remains. When chrony is fed from the hardware clock directly, the error bound it reports leaves out the hardware clock's own error. The Nitro system publishes that error separately, in nanoseconds, in the ENA driver's `phc_error_bound` file in sysfs, and the EC2 guide says to add it to the bound from chrony or ClockBound [42].

4. Latency Is a Distribution, Not a Number

The second and third questions have force because latency is a distribution, not a number. Every path has one, and it changes with what the system and the market are doing. Three of its properties matter more than any one figure: how tightly its tail, the rare slow observations above its high percentiles, is bounded; how it changes under load; and how it changes after a quiet spell. A fourth fact, about how races are won, decides what they are worth.

Determinism. Often the most valuable property of a latency distribution is how tightly its tail is bounded. Compare a path whose p50, its median, is 20 µs and whose p99.9 is 2 ms with one whose p50 is 40 µs and whose p99.9 is 60 µs (Figure 5). If a path is slowest on the updates that matter most, as the argument about load below suggests, the second path is the faster one where it counts. Without that link, two percentiles cannot settle it. Which path wins more races depends on the shape of both distributions and on where the rest of the field sits. Which path wins the races that pay depends on which updates its tail falls on. Part 7 takes up whether a path is slowest on the updates that matter most, and Part 8 what that is worth.

No general-purpose operating system offers latency fixed to the clock cycle, so in this series the word deterministic means something weaker. We call a path deterministic to the extent that its tail stays close to its median and its worst observed case stays within a stated limit, across days and across loads. Its measures are the spread, the difference between p99.9 and p50, and the maximum, with the number of samples and the conditions it was observed under. The series uses the difference rather than the ratio. A ratio suits tails that grow with the work, as queueing does. But the tails that destroy determinism are added by events whose length does not depend on the work, and the same added tail gives a smaller ratio over a longer median. A 500 µs pause makes the ratio of p99.9 to p50 26 over a 20 µs median but 6 over a 100 µs one, so a ratio would make the slower path look the more deterministic. A measurement shows an observed maximum, not a guarantee. On a general-purpose operating system, on a virtual machine and over TCP, rare events such as loss recovery on the order connection can exceed it at any time.

Determinism is lost to events that have nothing to do with the work the path is doing: page faults, scheduler pre-emption, interrupts landing on the wrong core, power-state transitions, memory allocation, hypervisor activity. Dean and Barroso show how, for a

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

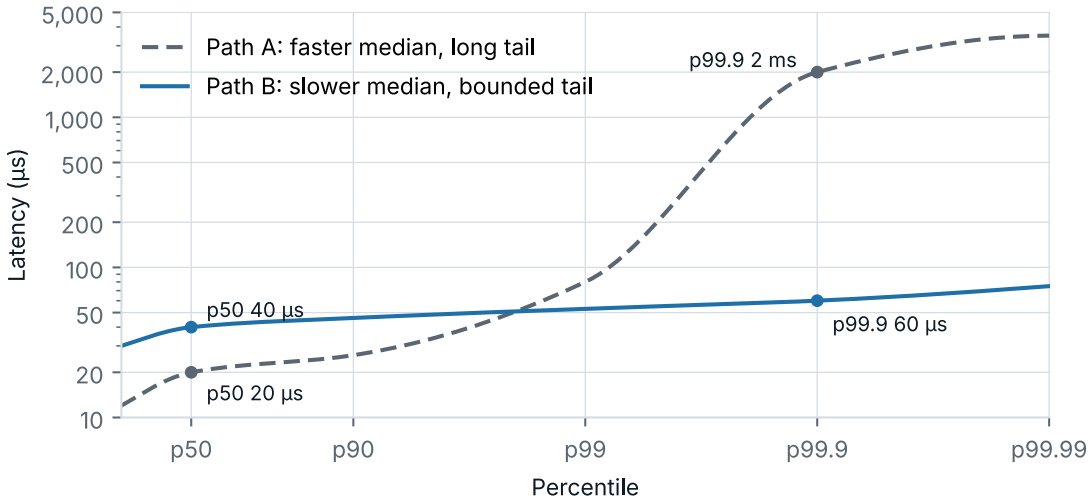


Figure 5: Two illustrative paths on a percentile plot. Percentile runs along the horizontal axis, stretched so that each further nine gets equal room, and latency is on a logarithmic vertical axis. Only the p50 and p99.9 of each path come from the text; the rest of each curve is an invented shape drawn through them, and in this drawing the curves cross near p98. Neither curve is a measurement. Source: figures/part-1-figures.py.

request fanned out to many servers, rare slow events at each one become common slow events for the whole [43]. A serial path has its own version of the same arithmetic. Call a pass slow when it takes longer than some time t, and a stage slow when it alone takes longer than t. Stage times add and none is negative, so a pass is slow whenever any of its stages is slow, and can be slow when none is. Its chance of a slow pass is therefore at least the chance that some stage is slow, and when such events are rare and seldom coincide, that chance is close to the sum of the stages' chances. Five stages each slow once in a thousand passes make a slow pass about once in two hundred.

Load. Market data does not arrive at an even rate, and we expect it to arrive fastest when prices move. A path that keeps up with the average rate can still queue during a burst, and queueing delay grows steeply as arrivals approach the rate the path can serve [44, Sec. 3.2]. Figure 6 illustrates this with a textbook model of a single stage, with Poisson arrivals and a fixed service time. For a given mean service time, a fixed one is the kindest case: variable service time raises the mean time in the system [44, Sec. 5.5], as more variable arrivals do [45]. Even so, the p99 time in the system is already about 4.3 times the service time at half load, about 11 times at 80 %, and about 23 times at 90 %. In heavy traffic the mean wait grows with the variance of the gaps between arrivals [45]. Gaps that bunch up are more variable than this model's random ones, so we expect market data to queue for longer than Figure 6 shows, and Part 7 will measure how much longer. The latency that matters is the latency during the busiest moments, and a figure averaged over a quiet session understates it.

Idle. The opposite also holds. After a quiet spell, a thread that sleeps between updates must wait for the scheduler to wake it, and its core may have entered a deeper idle state, whose worst-case exit latency the kernel lists for each state [47]. On EC2, only

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

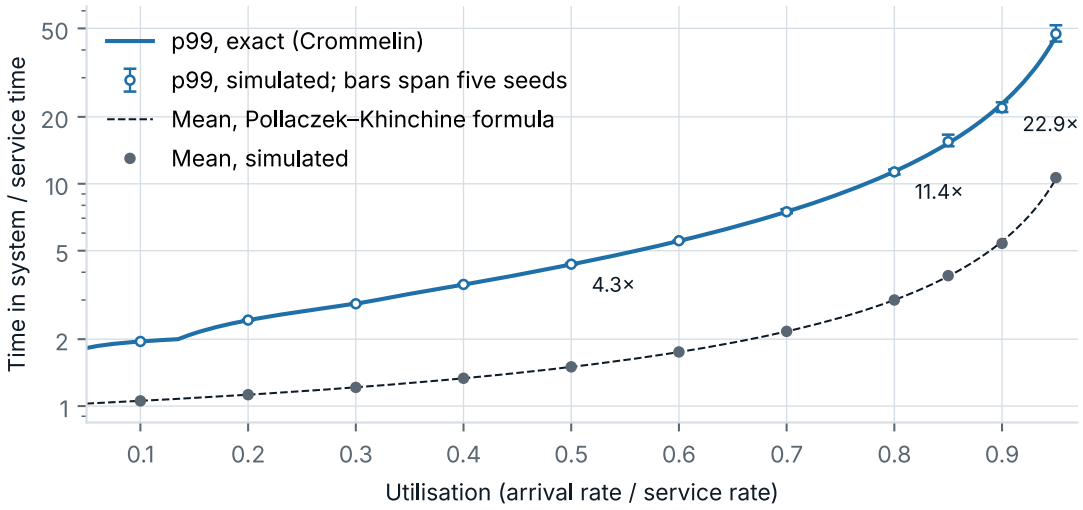


Figure 6: Time in a single-server queue with Poisson arrivals and a fixed service time, as a multiple of the service time on a logarithmic axis, against utilisation. The solid line is the exact p99, from Crommelin's waiting-time distribution [46]; its bend at twice the service time, near 13 % load, is exact, where the distribution changes form. The open points check it by simulation: five seeds of 400,000 arrivals per point, the first 40,000 of each discarded as warm-up. Each point's p99 is taken once from the five seeds' samples pooled, and its bars span the five seeds' separate p99s; below 80 % load the bars lie within the marker. The dashed line is the mean time in the system from the Pollaczek–Khinchine formula [44, Sec. 5.5], and the simulated means, the grey points, sit on it. Source: figures/part-1-figures.py.

some instance types let the guest control these states [48]. A thread that spins instead avoids both costs at the price of a core, the usual choice on a fast path. Even then, other work may have displaced the caches and branch predictors the path relies on, so the first update after the quiet can be slower than the hundredth.

The network stack also pays for idleness. Under the `tcp_slow_start_after_idle` setting, on by default [29], [30], a Linux sender that has sent nothing for longer than the retransmission timeout, by default at least 200 ms, halves the connection's congestion window for each full timeout of idleness, to no less than the initial window [33], [49]. Congestion controls that manage the window themselves, such as BBR, are exempt. On the firm's own order connection, only a burst of orders larger than the initial window is delayed: the excess waits at least a round trip for acknowledgements. A figure states the setting. If the venue's servers restart the window in the same way, the first burst of market data after a silence longer than their retransmission timeout is held back: the part beyond the initial window arrives at least a round trip late, and the firm cannot change that. A path's tail can therefore lengthen both at its busiest and just after it has been quiet.

Rank. For a market maker, latency pays off in races. In any single race, being one microsecond slower than the winner costs the same as being a hundred microseconds slower: the payoff is a step. Across many races, against competitors whose own latency varies and through a venue whose gateway adds variation of its own, the expected payoff is smoother, but it is still not a straight line in microseconds. An improvement is worth whatever it changes in the firm's rank among the participants racing on the

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

same footing. Rank also buys queue position, and on a venue that fills resting orders first come, first served, being earlier to the queue pays out smoothly over many fills. Part 8 will set out how to measure what added delay costs, by delaying a share of the engine's own decisions, and will publish the method, not the engine's result (Section 7). How that delay maps to rank, and what an improvement would be worth, follow only by inference, and Part 8 will say so.

These behaviours also put an instance's microseconds in proportion. The venue side of a cloud instance's round trip, including any batching of market data, the gateway and the matching engine, is outside the firm's control, and may be larger and more variable than anything on the instance. On-instance microseconds still pay in two places: in the tail during bursts, when a slow path turns a winnable race into a lost one, and in every race, to quote or to cancel, against the participants who reach the venue on the same footing, other cloud-hosted makers among them. What the firm controls is its own share: a short and, above all, tightly bounded path, so that when a race is lost, the loss is not in its own tail. That tail has a part of this series to itself, Part 6.

5. The Missing Samples

The second question is about the sample, and its best-known failure is coordinated omission: the act of measuring is coordinated with the system being measured, so that measurements are skipped exactly when the system is slow [2]. A load generator that sends on a schedule but waits for each response before the next send skips the slots that pass while it waits. A generator that keeps to its schedule queues the requests due during a stall, and the service takes them one after another once it ends. Figure 7 shows the effect for a generator sending 1,000 requests a second to a service that takes 50 µs, with one 150 ms stall in a 10-second run that begins as a request is sent. Percentiles here and throughout are Hyndman and Fan's type 1, which Section 6 defines. The generator that waits records 9,850 requests instead of 10,000, and the 150 slots that passed during the stall are the omission. Its p99 and p99.9 are both 50 µs, and the stall appears once, as its maximum. The generator that keeps to its schedule records all 10,000, each measured from the time it was due. Its p99 is 55.05 ms and its p99.9 is 140.55 ms. The maximum is 150.05 ms in both runs, which is why the maximum is the first figure to keep.

An engine that stamps every message it handles has not escaped the second question. Its sample can still go wrong in three ways, and its timing in a fourth. The simplest is counting only the updates that led to an order, which Section 2 answers with tick-to-decision. The other two are the market-data handler's cousins of coordinated omission: stamping an update when it was read rather than when it arrived, and missing the updates the engine never got to.

In the first cousin nothing is omitted, since every update is eventually read. But an update that arrives while an earlier one is being processed carries none of its queueing time, so a saturated path looks, from inside the application, like a fast one. The remedy is a stamp taken before the application is involved, at M0. That reduces the hidden queueing rather than removing it, because a kernel stamp follows the interface's inter-

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

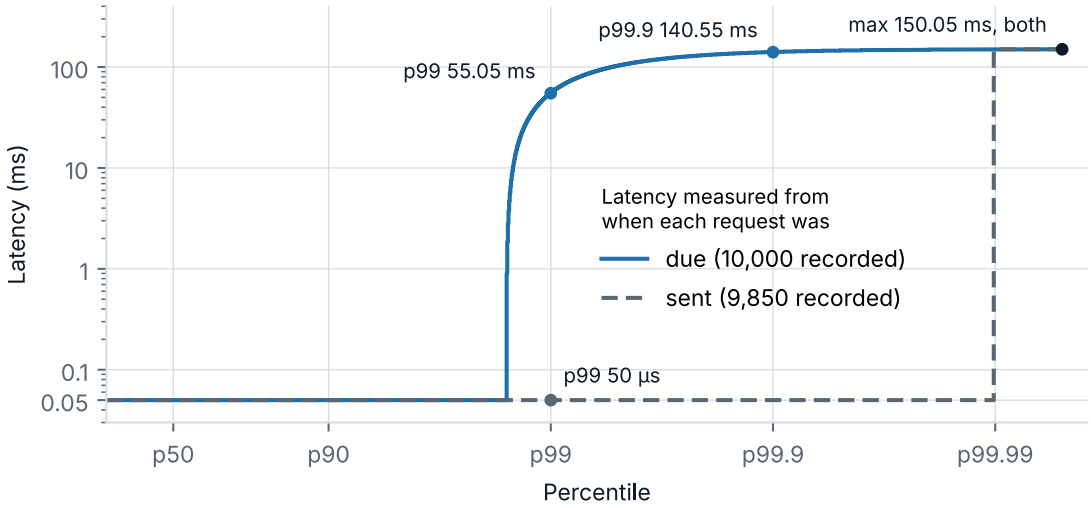


Figure 7: The same stall measured two ways: latency at each percentile for 1,000 requests a second to a 50 µs service, with one 150 ms stall in 10 seconds. The model is deterministic and every value shown is exact. The generator that waits for each response records 9,850 requests instead of 10,000 and hides the stall from every percentile up to p99.9; the generator that keeps to its schedule, measured from when each request was due, shows it from p99 upward. The dashed line steps at p99.99 because its one stalled sample in 9,850 is about one in 10,000. Percentile runs along the horizontal axis, stretched so that each further nine gets equal room; latency is on the vertical. The maximum has no place on a percentile axis and is marked at its right-hand end. Source: figures/part-1-figures.py.

rupt or poll, and, as Section 2 sets out, updates that share a packet share its stamp, and a read that returns several packets reports only one.

The second cousin is upstream of M0. On a TCP stream, when the application falls far enough behind, the socket's receive buffer fills, the receive window closes and the venue holds back the data. The queue has then moved to the venue's side of the connection, where no stamp on the instance can see it. A venue may instead drop a slow consumer's connection, which turns a latency problem into an availability one. With Linux's defaults the receive buffer starts at 128 KiB and autotuning enlarges it, up to 6 MiB on Linux 6.12, only after the application has read a burst quickly [20], [30]. Filling it therefore takes a stall of anything from a fraction of a second of the feed to several seconds, depending on the connection's history. A shorter stall leaves the data waiting on the host, where a kernel stamp does see it.

Loss hides part of the wait and shows the rest as inflated ingress. TCP delivers bytes in order, so a single lost segment on the market-data connection holds back every update behind it until it is recovered [50]. The stamp those updates then carry depends on where the kernel merged their segments before the read returned [16], [18], [20], and Part 3 will give the rule. Four checks expose these upstream queues and losses.

- The socket's queue depth, receive window and current receive-buffer size, and the TLS library's own plaintext buffer between the socket and the parser, which neither the socket nor `TCP_INFO` can see.
- The connection's counters, which Linux reports through `TCP_INFO` [29], [51], read by direction. The retransmission counters belong to the local sender and say little about

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

a market-data stream that is almost entirely receive. There, the out-of-order packet count, `tcpi_rcv_ooopack`, counts segments that arrive beyond a gap, a sign of loss or reordering, though a loss at the tail of a burst that is repaired before any later data arrives leaves no trace in it.

- The network interface's own counters. On EC2 the interface queues or drops packets when the instance exceeds its packet-rate or bandwidth allowances, and drops them when it exceeds its connection-tracking allowance, reporting them in the `bw_in_allowance_exceeded`, `bw_out_allowance_exceeded`, `pps_allowance_exceeded` and `conntrack_allowance_exceeded` counters, read with `ethtool -S` [52].
- The gap between M0 and the venue's own timestamp on each update. The venue does not publish its clock's error, and its timestamps may be coarser than the queueing they are meant to reveal, so the gap cannot give an absolute latency or measure the venue's share of the round trip. But a gap that widens during bursts shows data waiting upstream.

The fourth failure is timing. A figure from a quiet hour says nothing about a burst, so every figure is reported over the whole session and over its busiest minute: the one-minute window with the most updates by the venue's own timestamps. The window is chosen by the venue's count, never by latency or by arrivals at M0, which a closed receive window can shift, so that the figure is not conditioned on its own outcome. Part 7 will show how the busiest minute's distribution departs from the session's.

Keep the maximum, and report it with the number of samples and the conditions it was observed under. A maximum far beyond the p99.9 says that something happened which the percentiles do not describe: queueing the sample hid, or one of the events that destroy determinism, such as a page fault, a pre-emption or loss recovery on the order connection. The maximum also grows with the length of the run, so it means little without its count. A p99.9 drawn from a few thousand samples rests on its few largest observations. From 2,000, under the definition Section 6 names, it is the third largest, with at most two above it, so comparisons between runs need confidence intervals, not single figures. Slow observations also arrive in bursts rather than independently, so those intervals come from repeated runs or blocks of time, not from treating every sample as independent. Tene's advice is to ask, of anyone not measuring and plotting the maximum, what they are hiding from [2].

6. The Averaged Percentile

The third question is arithmetic. A percentile is a property of a sample: under any of the standard definitions, the p99 is computed from the observations themselves [53]. Percentiles cannot be averaged [2]. From twenty-four hourly p99s alone, all that can be said is that the daily p99 lies somewhere between the smallest and the largest of them (Figure 8). No more than 1 % of each hour's observations exceed that hour's p99, so no more than 1 % of the day's exceed the largest hourly p99. And more than 1 % of each hour's observations are at or above that hour's p99, so fewer than 99 % of the day's are below the smallest. The bound holds exactly for the definition this series uses, Hyndman and Fan's type 1: the smallest observation with at least that share of the

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

sample at or below it [53]. Under the interpolating and nearest-observation definitions the daily p99 can fall just outside that range, past one or more of the day's observations.

Nor can percentiles be combined across threads, machines or stages. The p99 of the whole path is not the sum of the p99s of its stages, nor bounded by it. Two stages each slow on 0.6 % of passes both have a p99 that misses the slow case, while the path, slow on about 1.2 %, does not.

Averaged percentiles are common because metric stores often keep a few percentiles per interval and discard the samples, after which only the wrong operations remain. Prometheus's documentation, for example, warns that the quantiles its summaries compute cannot be aggregated [54]. The remedy is to keep the distribution. Histograms with the same bucket layout, kept per interval, per thread and per stage, merge exactly. The merged histogram is the histogram of the union, and its percentiles are those of all the observations, to within the histogram's precision. The HdrHistogram design does this in a fixed amount of memory with a configurable relative precision, and its histograms can be added together [55]. The rule is then simple. Record into histograms, merge histograms, and compute percentiles once, from the merged whole.

Two smaller traps sit beside this one. The mean of a latency distribution is moved by a handful of samples in its tail, so a mean without its distribution says little. And a standard deviation is often read as if the distribution were normal, as in "within three

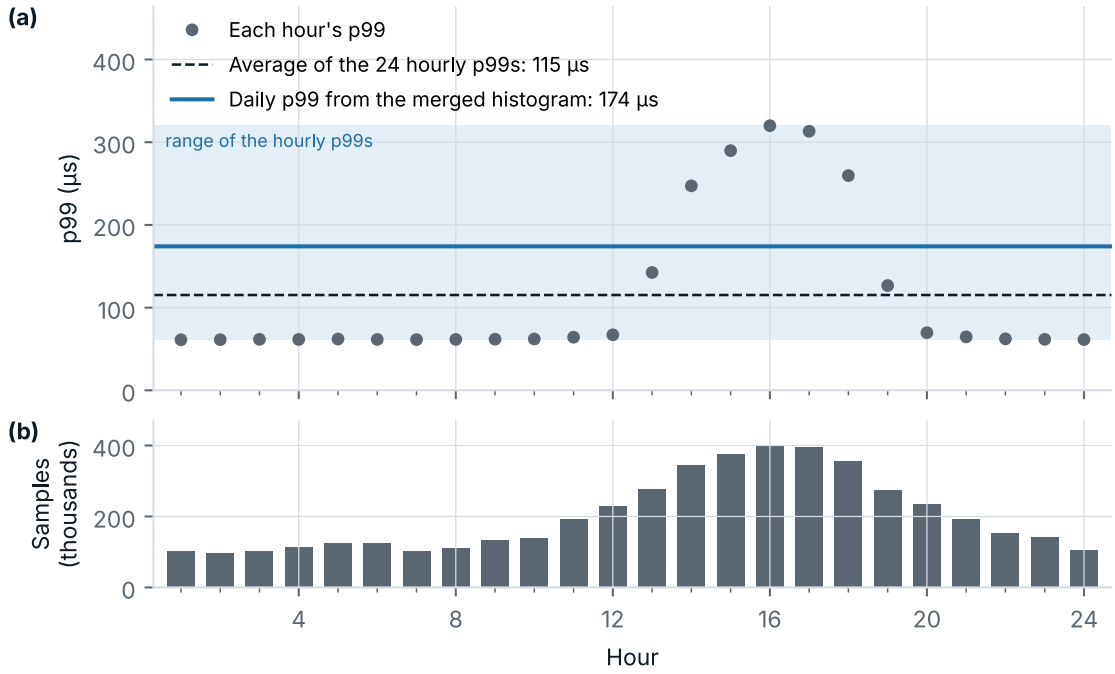


Figure 8: Averaging hourly p99s against merging hourly histograms, on synthetic data (seeded; not a measurement of the engine): 4,828,000 latencies over 24 hours, the busier hours having more samples and a longer tail. (a) Each hour's p99, the average of the 24, 115 µs, and the daily p99 taken once from the merged histogram, 174 µs. The band is the range of the hourly p99s, 61 µs to 320 µs, which must contain the daily p99. The histograms share one bucket layout with a relative precision of 1/128; the exact p99 of all the samples is 173 µs. All percentiles are Hyndman and Fan's type 1. (b) Samples per hour: the average gives each hour equal weight, whatever its count. Source: figures/part-1-figures.py.

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

sigma". Latency distributions are skewed and often have several modes, so that reading is wrong even when the standard deviation itself is computed correctly [2].

7. What This Series Measures

The subject of this series is Darkwave's market-making engine. Darkwave is built to make markets in crypto perpetuals with its own capital, and sells its engineering as Velocity Edge, so the engine and this series come from the same firm. The engine is written in Rust for Bybit and runs from a cloud instance in the venue's region. Each part states the instance class and the environment its figures came from. Many readers run beside the venue instead, in a co-location site, and the fastest run their path in hardware, so each part also sets out the method for a co-located host and for a path in hardware. Each part's figures come from running its method on the engine, its host class or the venue's public market data before the part is written. Its co-located and hardware methods are drawn from cited sources, and it names the measurement that would confirm each. The table sets out what is published and what is withheld.

Published	Withheld
The method, and every figure the path produces, but for the levels withheld below: all five stages and the whole path, at every percentile and the maximum, each with the conditions it was observed under, stated as market conditions, such as message rate relative to the session's median, never as instruments, times or absolute rates; tick-to-decision, tick-to-trade and tick-to-cancel with their sample counts rounded to an order of magnitude; loop health; the cost of measurement	Strategy and its parameters; the instruments quoted; exact counts of orders, cancels and messages per decision, and any figure split by instrument or by time of day; fill rates, markouts, profit and loss, and the results of Part 8's delay-injection experiment
The instance class, the kernel and driver, and the environment each figure came from, test or live; the region is stated	Placement: the zone and the position relative to the venue
The merged histograms behind every figure except those of the order round trip and the venue-timestamp gap, so a reader can take their own percentiles; those of tick-to-decision, tick-to-trade and tick-to-cancel are published normalised, as shares of their total, with the total rounded as above; beside every tick-to-trade figure, the connection's retransmission count, the shares of updates merged into a later decision and of updates never taken into one, and which stamp was used at each end; availability	The levels of the order round trip and of the gap between the venue's timestamps and M0, and their histograms, which together would locate the host

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Published	Withheld
For the order round trip and the venue-timestamp gap: the spread between p99.9 and p50, the maximum's excess over the median with its count, and the direction of each one's trend; for the order round trip these leave out retransmitted orders, reported as a count, because a retransmission's delay is a fixed timer plus the withheld round trip	—
The measurement crate and harness written for the series, as code, and snippets written for the paper; the host's operating-system settings that a figure depends on, as Parts 5 and 6 report them	Engine code and its configuration; hostnames, network addresses and accounts

The line is drawn where the edge is: the strategy, what it earns, and where the host sits. The host's figures, ingress and egress, come from a public instance class, kernel and driver, and Parts 2 to 6 will show anyone how to take the same measurements on the same class of host. The engine's own stages, decode, decide and encode, cannot be rerun without its code, which is withheld. They are published as observations with their apparatus, not as results a reader can reproduce, and what they give away is how fast the engine's code is, as any published benchmark does. A rival learns what it must beat, but not what the engine quotes, when, or from where. A taker who learns when the engine's cancel tail stretches cannot aim at it, because the venue's public data names no participant and the conditions are published as market conditions, not as times. Drawing the line any further in would make the series withhold the very figures it asks everyone else to show.

The series' unit of record is the nanosecond. Every stamp, histogram bucket boundary and stored figure is an integer count of nanoseconds. It is the unit the kernel's clocks and the hardware clock report in, and the one TSC readings are converted to. It is finer than any of them is accurate to, and finer than the 0.1 µs the regulation of Section 1 requires. The unit of presentation marks the boundary of the engine's own code. What that code does is reported in nanoseconds: decode, decide, encode and the cost of measurement. Whatever also passes through the operating system or the network is reported in microseconds, to a stated number of decimals: ingress, egress, the whole path, tick-to-decision, tick-to-cancel and the order round trip.

The parts are planned in this order, and further parts may follow the ninth. Any correction to a published part is issued as a new version with its revision history.

Part	Title	Question it answers
2	Which Clock Stamped It, and How Far to Trust It	Which clock stamps each milestone, what each costs to read, how the clocks are aligned, and how far they are from true time, measured on a cloud instance, and set out for a co-located host and a device

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Part	Title	Question it answers
3	Recording Without Slowing the Path	How to stamp every milestone, carry the application's, the kernel's and the hardware's stamps into the record, attribute each stamp to an update, and histogram every stage, with the cost of each probe against the stage it measures
4	Honest Summaries and Tracking That Never Stops	Percentiles from merged histograms, confidence on tails, drift, and alerts that fire on the right thing
5	Stamping at the Wire	Which stamp is closest to the truth: hardware stamps at M0 against the kernel's, and against the wire where the host is co-located or the path is in hardware; and the host, kernel and driver a figure must report so that its measurement can be repeated
6	Measuring Deterministic Latency	The spread between p99.9 and p50 and the maximum, with its count, across days and loads; working back from each slow event to its cause; what pinning, isolation and the host's other settings each remove from the tail, as measured effects; and what an observed bound can and cannot claim
7	Latency Under Load	Bursts, queueing and the quiet path: latency when the market moves and after it has been still; a test venue's traffic is not a live one's, and the part states which it measured
8	What Latency Costs	What a microsecond is worth: how fast the market makes a resting quote stale, measured from the venue's public data, and delay injection, a randomised experiment that holds back a share of quoting decisions; the method and the market's figures are published, the price is not
9	When the Path Is an FPGA	How the milestones, clocks, samples and summaries of this series apply when the path is logic on an FPGA rather than software on a host, set out from cited sources with the measurement that would confirm each, and what a wire-to-wire figure can and cannot be compared with

Every claim in the series that draws on outside work is cited to its source. Every figure the series produces itself comes with the script that produced it, or with its

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

measurement apparatus described well enough to rebuild. Where a claim is the series' own reasoning, it is argued in the text rather than asserted. And where none of these is possible, the paper says so and names the measurement that would settle the question.

Conclusion

The next time you are shown a latency figure, ask three questions. Which two events does it run between, which clocks stamped them, and how far were those clocks from true time and from each other? Which samples were taken, and were they stamped when they arrived or when they were read? How was it summarised, and are the histograms still there? Then ask for the p99.9 and the maximum beside the median, over the whole session and over its busiest minute, with the sample counts and the histograms behind them. This part carries no measurement of the engine; its figures are diagrams, illustrations and models. The series will answer these questions for its own engine, beginning with the clocks in Part 2.

Glossary

Term	Meaning
Availability	The share of the session during which the path was live
Clock error bound	The time daemon's estimate of how far the local clock may be from true time at a given moment
Coordinated omission	Skipping measurements because the system is slow, as a load generator does when it waits for each response before sending the next, so the sample loses its slowest cases and flatters the system
Cross-connect	A cable, provided by the site operator, between a firm's equipment and the venue's within a co-location site
Deterministic latency	In this series, latency whose tail stays close to its median and whose observed worst case stays within a stated limit, reported with its sample count and conditions; a weaker sense than the fixed cycle count the term means in hardware
Ingress, decode, decide, encode, egress	The five stages of tick-to-trade, M0 to M1, M1 to M2, M2 to M3, M3 to M4 and M4 to M5, as Section 2 defines them
Loop health	Whether the receiving thread was ready when an update arrived, or still busy with earlier ones
Markout	The difference between a fill's price and a reference price, usually the mid, a fixed time later, signed by the side of the fill; a measure of adverse selection
Milestone	An instant on the path at which a stamp is taken

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Term	Meaning
Order round trip	From the order's hand-off at M5 to the arrival of the venue's acknowledgement
Percentile	The pN of a sample, such as p50, p99 or p99.9: the smallest observation with at least N % of the sample at or below it, Hyndman and Fan's type 1, the definition this series uses throughout
Percentile plot	A distribution drawn with percentile along the horizontal axis, stretched so that p90, p99 and p99.9 are evenly spaced, and latency on the vertical; its tail is the rising right-hand end
Spread	The difference between a distribution's p99.9 and its p50, the series' measure of how tightly the tail is bounded
Stage	The interval between two consecutive milestones
Tail	The slow end of a latency distribution: the rare observations above its high percentiles. It lengthens when those percentiles and the maximum grow while the median holds
Tap	An inline device on a network link that copies its traffic to capture equipment outside the path
Tick-to-cancel	The interval from the M0 of an update that moved the book against a resting quote to the M5 of the cancel
Tick-to-decision	The interval from the arrival of an update to the decision that took it into account, measured for every update
Tick-to-trade	The interval from M0 to M5 for an update that leads to an order: from the update's arrival to the order's hand-off to the interface, as Section 2 defines each for a cloud instance, a co-located host and a path in hardware
TSC	The time-stamp counter, a register on each logical processor that runs at a constant rate on processors with the invariant TSC

References

[1] CryptoQuant, "FY 2025 review of crypto exchange activity." CryptoQuant Quicktake, Jan. 2026. Accessed: Oct. 18, 2026. [Online]. Available: https://cryptoquant.com/insights/quicktake/69650eb3cbe161236f138693-FY-2025-Review-of-Crypto-Exchange-Activity

[2] G. Tene, "How NOT to measure latency." QCon San Francisco, slides published by InfoQ, 2015. Accessed: Oct. 15, 2026. [Online]. Available: https://videog.infoq.com/downloads/pdfdownloads/presentations/QConSF2015-GilTene-HowNOTtomeasureLatency.pdf

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

[3] J. Hasbrouck and G. Saar, "Low-latency trading," *Journal of Financial Markets*, vol. 16, no. 4, pp. 646–679, 2013, doi: 10.1016/j.finmar.2013.05.003.

[4] E. Budish, P. Cramton, and J. Shim, "The high-frequency trading arms race: Frequent batch auctions as a market design response," *The Quarterly Journal of Economics*, vol. 130, no. 4, pp. 1547–1621, 2015, doi: 10.1093/qje/qjv027.

[5] M. Aquilina, E. Budish, and P. O'Neill, "Quantifying the high-frequency trading 'arms race'," *The Quarterly Journal of Economics*, vol. 137, no. 1, pp. 493–564, 2022, doi: 10.1093/qje/qjab032.

[6] European Commission, "Commission delegated regulation (EU) 2025/1155 of 12 June 2025 supplementing Regulation (EU) No 600/2014 with regard to regulatory technical standards specifying the input and output data of consolidated tapes, the synchronisation of business clocks and the revenue redistribution by the consolidated tape provider for shares and ETFs, and repealing Commission Delegated Regulation (EU) 2017/574." Official Journal of the European Union, L series, 2025/1155, 3 November, 2025. Accessed: Oct. 15, 2026. [Online]. Available: https://eur-lex.europa.eu/eli/reg_del/2025/1155/oj

[7] European Commission, "Commission delegated regulation (EU) 2017/574 of 7 June 2016 supplementing Directive 2014/65/EU with regard to regulatory technical standards for the level of accuracy of business clocks." Official Journal of the European Union, L 87, p. 148, 31 March, 2017. Accessed: Oct. 15, 2026. [Online]. Available: https://eur-lex.europa.eu/eli/reg_del/2017/574/oj

[8] STAC, "STAC-T0 overview." Securities Technology Analysis Center, Nov. 2020. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.stacresearch.com/system/files/central/STAC-T0%20backgrounder.pdf

[9] Bybit, "Connect." Bybit API documentation, v5. Accessed: Oct. 15, 2026. [Online]. Available: https://bybit-exchange.github.io/docs/v5/ws/connect

[10] Binance, "Spot WebSocket market streams." Binance Spot API documentation. Accessed: Oct. 15, 2026. [Online]. Available: https://developers.binance.com/docs/binance-spot-api-docs/web-socket-streams

[11] OKX, "OKX API guide." OKX API documentation. Accessed: Oct. 15, 2026. [Online]. Available: https://www.okx.com/docs-v5/en/

[12] I. Fette and A. Melnikov, "The WebSocket protocol." RFC 6455, 2011. doi: 10.17487/RFC6455.

[13] D. McGrew, "An interface and algorithms for authenticated encryption." RFC 5116, 2008. doi: 10.17487/RFC5116.

[14] E. Rescorla, "The transport layer security (TLS) protocol version 1.3." RFC 8446, obsoleted by RFC 9846, 2018. doi: 10.17487/RFC8446.

[15] E. Rescorla, "The transport layer security (TLS) protocol version 1.3." RFC 9846, obsoletes RFC 8446, Jul. 2026. Accessed: Oct. 24, 2026. [Online]. Available: https://www.rfc-editor.org/info/rfc9846

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

[16] Linux kernel contributors, “net/ipv4/tcp.c: tcp_should_autocork, tcp_recvmsg_locked.” Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/net/ipv4/tcp.c

[17] Linux kernel contributors, “Segmentation offloads.” Linux kernel documentation, networking. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html

[18] Linux kernel contributors, “net/core/dev.c: net_timestamp_check (macro), netif_receive_skb_list_internal.” Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/net/core/dev.c

[19] Linux kernel contributors, “net/core/gro.c: skb_gro_receive.” Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/net/core/gro.c

[20] Linux kernel contributors, “net/ipv4/tcp_input.c: tcp_try_coalesce, tcp_rcv_space_adjust.” Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/net/ipv4/tcp_input.c

[21] Linux kernel contributors, “Linux kernel driver for Elastic Network Adapter (ENA) family.” Linux kernel documentation, networking device drivers. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/device_drivers/ethernet/amazon/ena.html

[22] Amazon Web Services, “Precision clock and time synchronization on your EC2 instance.” Amazon EC2 User Guide. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html

[23] Amazon Web Services, “Set the time reference on your EC2 instance to use the local Amazon Time Sync Service.” Amazon EC2 User Guide. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/configure-ec2-ntp.html

[24] Amazon Web Services, “README for the Linux ENA driver.” amzn-drivers repository, kernel/linux/ena/README.rst. Accessed: Oct. 24, 2026. [Online]. Available: https://github.com/amzn/amzn-drivers/blob/master/kernel/linux/ena/README.rst

[25] Linux kernel contributors, “Timestamping.” Linux kernel documentation, networking. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/timestamping.html

[26] Bybit, “Orderbook.” Bybit V5 API documentation. Accessed: Oct. 18, 2026. [Online]. Available: https://bybit-exchange.github.io/docs/v5/websocket/public/orderbook

[27] J. Nagle, “Congestion control in IP/TCP internetworks.” RFC 896, 1984. doi: 10.17487/RFC0896.

[28] R. Braden, Ed., “Requirements for internet hosts – communication layers.” RFC 1122, 1989. doi: 10.17487/RFC1122.

[29] Linux man-pages project, “tcp(7).” Linux man-pages 6.19. Accessed: Oct. 15, 2026. [Online]. Available: https://man7.org/linux/man-pages/man7/tcp.7.html

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

[30] Linux kernel contributors, “IP sysctl.” Linux kernel documentation, networking. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/networking/ip-sysctl.html

[31] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, “The RACK-TLP loss detection algorithm for TCP.” RFC 8985, 2021. doi: 10.17487/RFC8985.

[32] V. Paxson, M. Allman, J. Chu, and M. Sargent, “Computing TCP’s retransmission timer.” RFC 6298, 2011. doi: 10.17487/RFC6298.

[33] Linux kernel contributors, “net/ipv4/tcp_output.c: tcp_schedule_loss_probe, tcp_cwnd_restart.” Linux kernel source, v6.12. Accessed: Oct. 18, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/net/ipv4/tcp_output.c

[34] Linux kernel contributors, “drivers/net/ethernet/amazon/ena/ena_netdev.c: ena_start_xmit.” Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/drivers/net/ethernet/amazon/ena/ena_netdev.c

[35] Linux man-pages project, “clock_gettime(2).” Linux man-pages 6.19. Accessed: Oct. 15, 2026. [Online]. Available: https://man7.org/linux/man-pages/man2/clock_gettime.2.html

[36] Intel Corporation, “Intel 64 and IA-32 architectures software developer’s manual.” Volume 3B, chapter 18, “Debug, Branch Profile, TSC, and Intel Resource Director Technology (Intel RDT) Features,” section 18.17.1, “Invariant TSC.” Accessed: Oct. 24, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[37] Linux kernel contributors, “KVM-specific MSRs.” Linux kernel documentation, KVM. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/virt/kvm/x86/msr.html

[38] Linux kernel contributors, “include/linux/skbuff.h: __net_timestamp.” Linux kernel source, v6.12. Accessed: Oct. 24, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/include/linux/skbuff.h

[39] chrony project, “chrony.conf(5).” chrony documentation. Accessed: Oct. 15, 2026. [Online]. Available: https://chrony-project.org/doc/latest/chrony.conf.html

[40] J. Levinson and J. Ridoux, “It’s about time: Microsecond-accurate clocks on Amazon EC2 instances.” AWS Compute Blog, Nov. 2023. Accessed: Oct. 15, 2026. [Online]. Available: https://aws.amazon.com/blogs/compute/its-about-time-microsecond-accurate-clocks-on-amazon-ec2-instances/

[41] Amazon Web Services, “ClockBound.” GitHub repository. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/aws/clock-bound

[42] Amazon Web Services, “Compare timestamps for your Linux instances.” Amazon EC2 User Guide. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compare-timestamps-with-clockbound.html

[43] J. Dean and L. A. Barroso, “The tail at scale,” *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013, doi: 10.1145/2408776.2408794.

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

[44] L. Kleinrock, *Queueing systems, volume 1: theory*. New York: Wiley, 1975.

[45] J. F. C. Kingman, "The single server queue in heavy traffic," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 57, no. 4, pp. 902–904, 1961, doi: 10.1017/S0305004100036094.

[46] C. D. Crommelin, "Delay probability formulae when the holding times are constant," *Post Office Electrical Engineers' Journal*, vol. 25, pp. 41–50, 1932.

[47] Linux kernel contributors, "CPU idle time management." Linux kernel documentation, admin guide. Accessed: Oct. 15, 2026. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/pm/cpuidle.html

[48] Amazon Web Services, "Processor state control for Amazon EC2 Linux instances." Amazon EC2 User Guide. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/processor_state_control.html

[49] Linux kernel contributors, "include/net/tcp.h: tcp_slow_start_after_idle_check." Linux kernel source, v6.12. Accessed: Oct. 24, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/include/net/tcp.h

[50] W. Eddy, Ed., "Transmission control protocol (TCP)." RFC 9293, 2022. doi: 10.17487/RFC9293.

[51] Linux kernel contributors, "include/uapi/linux/tcp.h: struct tcp_info." Linux kernel source, v6.12. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/torvalds/linux/blob/v6.12/include/uapi/linux/tcp.h

[52] Amazon Web Services, "Monitor network performance for ENA settings on your EC2 instance." Amazon EC2 User Guide. Accessed: Oct. 15, 2026. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring-network-performance-ena.html

[53] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, vol. 50, no. 4, pp. 361–365, 1996, doi: 10.1080/00031305.1996.10473566.

[54] Prometheus Authors, "Histograms and summaries." Prometheus documentation. Accessed: Oct. 15, 2026. [Online]. Available: https://prometheus.io/docs/practices/histograms/

[55] HdrHistogram contributors, "HdrHistogram: A high dynamic range (HDR) histogram." GitHub repository. Accessed: Oct. 15, 2026. [Online]. Available: https://github.com/HdrHistogram/HdrHistogram

Revision History

Version	Date	Change
1.0	2026-10-24	First publication
1.0.0	2026-11-01	Style revision: prose tightened for readability; no claim, figure, definition or citation changed

© 2026 Darkwave Trading Limited, trading as Velocity Edge · CC BY 4.0

Licence and Disclaimer

© 2026 Darkwave Trading Limited, trading as Velocity Edge. This paper is licensed under the Creative Commons Attribution 4.0 International licence (CC BY 4.0, https://creativecommons.org/licenses/by/4.0/): you may share and adapt it for any purpose, provided you credit Phillip Pienaar and Velocity Edge, keep this copyright notice, link to the licence, link to the original where reasonably practicable, and indicate any changes. The Darkwave and Velocity Edge names and logos are not covered by this licence.

This paper is for education and information. It is not investment advice, and nothing in it is a recommendation to trade.