UNIVERSITY OF
THESSALY

SCHOOL OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE & TELECOMMUNICATIONS

Synchronization of Virtualized Network Elements for Multi-Domain Networks Beyond 5G

by

Nikolaos Boukouvalas

FINAL THESIS

ADVISOR

Foukalas Fotios
Assistant Professor

Lamia, 30th June, 2025

«Με ατομική μου ευθύνη και γνωρίζοντας τις κυρώσεις [(1)], που προβλέπονται από της διατάξεις της παρ. 6 του άρθρου 22 του Ν. 1599/1986, δηλώνω ότι:

*1. Δεν παραθέτω κομμάτια βιβλίων ή άρθρων ή εργασιών άλλων αυτολεξεί **χωρίς να τα περικλείω σε εισαγωγικά** και χωρίς να αναφέρω το συγγραφέα, τη χρονολογία, τη σελίδα. Η αυτολεξεί παράθεση χωρίς εισαγωγικά χωρίς αναφορά στην πηγή, είναι λογοκλοπή. Πέραν της αυτολεξεί παράθεσης, λογοκλοπή θεωρείται και η παράφραση εδαφίων από έργα άλλων, συμπεριλαμβανομένων και έργων συμφοιτητών μου, καθώς και η παράθεση στοιχείων που άλλοι συνέλεξαν ή επεξεργάσθηκαν, χωρίς αναφορά στην πηγή. Αναφέρω πάντοτε με πληρότητα την πηγή κάτω από τον πίνακα ή σχέδιο, όπως στα παραθέματα.*

*2. Δέχομαι ότι η αυτολεξεί **παράθεση χωρίς εισαγωγικά**, ακόμα κι αν συνοδεύεται από αναφορά στην πηγή σε κάποιο άλλο σημείο του κειμένου ή στο τέλος του, είναι αντιγραφή. Η αναφορά στην πηγή στο τέλος π.χ. μιας παραγράφου ή μιας σελίδας, δεν δικαιολογεί συρραφή εδαφίων έργου άλλου συγγραφέα, έστω και παραφρασμένων, και παρουσίασή τους ως δική μου εργασία.*

3. Δέχομαι ότι υπάρχει επίσης περιορισμός στο μέγεθος και στη συχνότητα των παραθεμάτων που μπορώ να εντάξω στην εργασία μου εντός εισαγωγικών. Κάθε μεγάλο παράθεμα (π.χ. σε πίνακα ή πλαίσιο, κλπ), προϋποθέτει ειδικές ρυθμίσεις, και όταν δημοσιεύεται προϋποθέτει την άδεια του συγγραφέα ή του εκδότη. Το ίδιο και οι πίνακες και τα σχέδια

4. Δέχομαι όλες τις συνέπειες σε περίπτωση λογοκλοπής ή αντιγραφής.

Ημερομηνία: 30/6/2025

Ο Δηλών

(1) «Όποιος εν γνώσει του δηλώνει ψευδή γεγονότα ή αρνείται ή αποκρύπτει τα αληθινά με έγγραφη υπεύθυνη δήλωση
του άρθρου 8 παρ. 4 Ν. 1599/1986 τιμωρείται με φυλάκιση τουλάχιστον τριών μηνών. Εάν ο υπαίτιος αυτών των πράξεων
σκόπευε να προσπορίσει στον εαυτόν του ή σε άλλον περιουσιακό όφελος βλάπτοντας τρίτον ή σκόπευε να βλάψει άλλον, τιμωρείται με κάθειρξη μέχρι 10 ετών.»

Περιεχόμενα

1. **Introduction**

The evolution of mobile communications has reached a point where architectural rigidity and legacy constraints no longer suffice. In the fifth generation (5G) of cellular networks and beyond, expectations for ultra-low latency, high reliability, and dense device connectivity are rewriting the rules of deployment. Traditional monolithic infrastructures however robust struggle to cope with the emerging complexity. Something had to change.

Virtualization has emerged not merely as a solution, but as a foundational shift. Technologies like Docker [1] offer the promise of modular, lightweight, and scalable components that can be orchestrated with far greater flexibility than their hardware-bound predecessors. This thesis explores the practical integration of Open5GS, an open-source 5G core network, with UERANSIM [2], a simulation platform for gNodeBs (gNBs) and User Equipments (UEs), in a dockerized environment. The approach centers not only on deployment architecture but also on the intricate problem of time synchronization, which is vital when coordinating distributed elements in a virtualized radio access network.

Accurate timing becomes particularly critical in systems relying on Time Division Duplexing (TDD) and Coordinated Multi-Point (CoMP) techniques. In such contexts, even minor deviations in clock alignment between network elements can cascade into degraded performance, disrupted sessions, and loss of service continuity. To mitigate this, the IEEE 1588 Precision Time Protocol (PTP) is employed as a synchronization mechanism. Yet PTP, elegant though it is, can falter in noisy, packet-switched environments. Software timestamping while more flexible than hardware-based alternatives introduces its own imperfections, namely in the form of variable latency and offset drift.

This thesis proposes a hybrid solution. Rather than embedding timing correction directly within the PTP source code which, for reasons of maintainability and separation of concerns, might not always be practical a Kalman Filter is implemented as a separate module. It runs in its own Docker container, continuously parsing logged synchronization metrics and dynamically adjusting the master offset. While not tightly coupled to the PTP stack, this approach

enables rapid iteration, easier debugging, and most importantly measurable improvements in synchronization stability.

The investigation proceeds through a detailed analysis of the dockerized 5G core and RAN components, followed by a focused examination of the PTP-Kalman integration. Experimental results are drawn from actual system logs, not synthetic models. They tell a story of progress but also of limitation. The Kalman Filter improves timing precision; that much is certain. Yet the degree of improvement, as this work will show, is bounded by practical factors: packet delay variation (PDV), imperfect models, and the inherent messiness of real-world systems.

Still, this is a step forward. A modular, dockerized 5G network architecture, enhanced by algorithmic time compensation, holds real promise for scalable, resilient telecommunications infrastructure. And although the road to perfection remains long, every microsecond reclaimed is, in a way, a victory.

2. Virtualized 5G Core Network Using Docker Technology

There was a time when building a mobile core network meant racks of hardware, custom firmware, and weeks of provisioning. And maybe, for large operators, that world still exists. But in academic and research settings, such scale is neither feasible nor desirable. What's needed is agility. The freedom to tear down, reconfigure, and rebuild within minutes not hours or days. And Docker, perhaps reluctantly at first, answered that need.

In this work, the entire 5G core network comprising the Access and Mobility Management Function (AMF), Session Management Function (SMF), User Plane Function (UPF), along with supporting modules like the NRF, PCF, and AUSF was deployed using Open5GS, all within a containerized environment. The choice wasn't arbitrary. Open5GS aligns with ETSI specifications outlined in TS 123 501 [3], ensuring 3GPP-compliant interfaces and modularity. More importantly, it works well with Docker, requiring only a few network bindings and volume mounts to become fully operational [4].

Each component, while designed to run independently, was grouped into a single container for this study. It may sound counterintuitive. Why compress a disaggregated system into one package? The reason was pragmatic: reduced complexity. By deploying all functions together, we could ensure deterministic behavior in inter-service communications, while avoiding the noise introduced by multiple containers competing for network and CPU resources. It’s a simplification but a revealing one.

Once launched, the container behaves as expected. Services initiate in sequence, and the core begins listening for gNodeB attachments. Network interfaces are exposed to a Docker bridge network, ensuring connectivity between components. Logs begin to populate. The illusion of a fully functional 5GC though confined to a single Linux namespace becomes surprisingly convincing.

Yet, issues soon emerge. Docker, by default, abstracts many hardware behaviors, including timekeeping. And since each service within the container communicates using UDP or SCTP, packet order and latency become more than implementation details they become symptoms. During some runs, authentication delays increased inexplicably. A deeper inspection of packet traces suggested nothing dramatic just slight fluctuations in response intervals. But in real-time systems, such fluctuations aren’t benign.

To better understand these patterns, the host’s CPU scheduler behavior was examined. When multiple threads from different network functions attempted to access shared memory or disk I/O simultaneously, Docker’s container runtime queued their requests with no knowledge of their latency sensitivity. This led to a realization: Docker's lightweight nature is both its strength and its Achilles’ heel.

The ease of deployment was undeniable. With one command, the entire 5GC could be spun up in seconds. But with that speed came opacity. It became difficult to know where performance bottlenecks were rooted. Was it the AMF’s processing logic? Or the host kernel delaying packet forwarding? Or perhaps Docker itself juggling cgroups, namespaces, and virtual NICs without any awareness of the timing stakes involved?

This ambiguity, while frustrating, was oddly productive. It forced a more careful approach to measurement. Rather than assume determinism, the system had to be probed, logged, and interpreted slowly. And in doing so, the quirks of a containerized mobile core began to surface not as bugs, but as characteristics of a fundamentally virtual architecture.

Some may argue that deploying Open5GS as a monolithic container undermines the principles of network function disaggregation. And they would not be wrong. But within the scope of this experiment, simplification served its purpose. It allowed full control over timing, resource usage, and error propagation creating a kind of laboratory environment, where container behaviors could be studied in isolation before scaling outward.

In short, Docker didn't make things perfect. It made them visible.

2.1 Virtualized Access with UERANSIM: Emulating the gNodeB

If the 5G core forms the brain of the network, the gNodeB acts as its sensing and movement system. It connects users, monitors channels, negotiates timing. And while in this study no physical radio transmission occurred, the illusion of radio behavior had to be convincing enough for the core to respond as if the signals were real.

This is where UERANSIM comes in. A lightweight, open-source project, UERANSIM simulates both User Equipment (UE) and gNodeB nodes, communicating via NGAP and GTP-U, just like their real-world counterparts. For the 5GC, there's no meaningful difference if the packets arrive on time and with the expected structure, the origin might as well be a physical tower on a real hilltop [5].

Dockerizing UERANSIM was relatively straightforward. Each simulated gNB runs in its own container, configured with access to the same bridge network as the core. Similarly, UE instances are launched as sibling containers. They send connection requests, receive session parameters, and even exchange data-plane packets with the UPF. From a network standpoint, the system behaves realistically. But from a temporal perspective, the cracks start to show [6].

The first sign of trouble was randomness. One test run would complete smoothly. Another, with identical parameters, would show dropped connections or attach failures. The logs offered clues timeouts, unacknowledged responses but no smoking gun. Over time, a pattern emerged: these failures correlated with changes in CPU load on the host. More specifically, they often followed the launch of a new container [6].

Unlike physical RAN elements, UERANSIM doesn't operate with access to hardware clocks. It timestamps messages using the host kernel's perception of time. And that kernel, burdened by multiple Docker threads and virtual interfaces, sometimes stumbles. A few milliseconds here, a hundred microseconds there. Not much. Until it is.

Consider the NG Setup procedure. When the gNB contacts the AMF to register itself, it expects a response within a short window. If the response is delayed perhaps because the AMF was momentarily deprived of CPU the gNB assumes failure. It retries. And if that retry arrives before the AMF has recovered, the session collapses.

This is not a bug. It is a timing mismatch a subtle dance where both partners keep stepping on each other's feet [7]. Software-based approaches like TimeKeeper have attempted to address these challenges without specialized hardware [8], but they still face many of the same inconsistencies observed here.

To address this, experiments were run using CPU pinning and Docker resource limits. The gNB was assigned exclusive access to a dedicated core, isolated from the Open5GS container. This reduced but did not eliminate the anomalies. Meanwhile, changes to Docker's networking mode were explored. Host networking improved timing precision slightly but broke namespace separation, making UE-to-UPF routing difficult. Macvlan offered promise, but its configuration proved brittle across reboots [6]. In the end, simplicity prevailed: a shared bridge network, monitored carefully.

Interestingly, running UERANSIM outside Docker directly on the host reduced jitter significantly. But doing so sacrificed containment, making it harder to reset the environment between

runs. Once again, containerization offered trade-offs: speed and control, at the cost of precision.

And yet, the results were usable. Despite imperfections, most UEs registered successfully. GTP tunnels were established. Data flowed. Logs confirmed the core reacted to simulated user activity. And the illusion though not perfect held.

There's something oddly satisfying about watching ten UE containers light up in sequence, like actors entering a stage. They ask to connect. The network replies. Even knowing it's all simulated doesn't diminish the impact. It works. Or at least, it almost does.

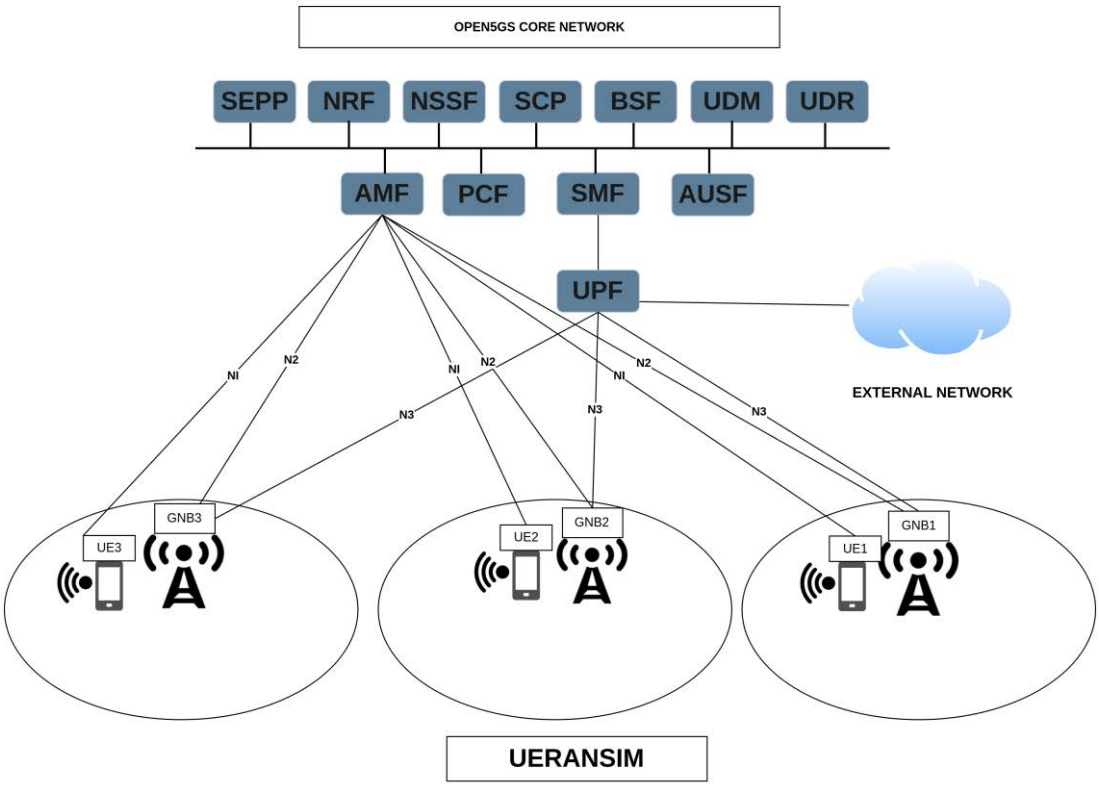

Figure 1 OPEN5GS/RAN Architecture [17]

2.2 Synchronization in Containerized 5G Deployments: Time, Drift, and the Role of PTP

Time, in a 5G network, is more than a background detail. It governs when packets are sent, how handovers are triggered, and whether latency-sensitive services operate at all. And yet, in a containerized environment, time is fuzzy. It drifts. It stalls. It plays tricks. You look at the logs and something feels off. A packet sent at 14:02:11 arrives somehow before its predecessor timestamped 14:02:12. It shouldn't happen. But it does.

In physical setups, this problem is addressed through hardware timestamping, often embedded directly into network interface cards (NICs) that support the IEEE 1588 Precision Time Protocol (PTP). But in this research, the environment was entirely virtualized no specialized hardware, no physical clocks. Just software.

So PTP was brought in anyway. Not for perfection, but for structure. The goal wasn't nanosecond accuracy. It was predictability, or at least, measurable deviation. The setup relied on ptp4l and phc2sys, running inside privileged containers. One acted as the master clock; the others attempted to synchronize. Offsets were logged every second. Deviations of 200–400 microseconds were common. Occasionally, a spike would appear a sudden drift up to 800 μs or more usually after a container restart or CPU reallocation [9].

The behavior wasn't linear. Some offsets shrank with time, others grew. A few held steady before lurching sideways. This isn't unusual in software-based PTP [10]. Kernel scheduling, TCP/IP stack delays, and the unpredictable nature of Docker's network bridges all contribute. Containers don't exist in a vacuum they share the host's hardware with everything else. And that sharing introduces noise.

It became clear that software timestamping was both the problem and the workaround. On one hand, it caused most of the drift. On the other, it enabled drift measurement in the first place. And with those measurements, came a new possibility: filtering.

That's where the Kalman filter discussed more thoroughly in Section 3 entered the picture. But even before any correction was applied, the raw PTP logs told a story. They showed when

the network was calm, when it was under load, and when background processes distorted the synchronization loop. They revealed patterns in what first appeared as randomness.

It's important to acknowledge: this is not how PTP was meant to be used. Without hardware support, many of its promises are undermined. Sub-microsecond precision becomes, optimistically, low-microsecond consistency. But for this experiment, that was enough.

The offset plots, when visualized, created a kind of heartbeat. Not perfectly regular, but not chaotic either. Just organic. The system was alive, pulsing with uncertainty.

At one point, the experiment was paused for several hours, then resumed. The offset drifted rapidly over 1000 μs before slowly settling again. No restart had occurred. Just idle time. Maybe Docker rebalanced threads. Maybe the host's NTP daemon introduced interference [11]. The logs didn't explain it. But they hinted.

Somewhere in all this, it became obvious: time inside a container is negotiable. Not unreliable, exactly but contingent. Dependent on how many other things are asking for attention. Dependent on heat, interrupts, kernel versions.

Synchronization in this environment isn't a matter of configuration it's a conversation between competing processes. And sometimes, they don't agree.

2.3 Reflections on Dockerized Architectures for 5G: Strengths and Structural Fault Lines

Docker gives a lot, quickly. That's part of its appeal. A single command brings up an entire 5G core. No manual configuration of system services. No worrying about kernel modules or init scripts. You don't even need to think about network interfaces Docker handles that too. This convenience, however, is never free.

In this study, the containerization of Open5GS and UERANSIM allowed rapid deployment of a functional 5G network. Each function, neatly packed into its own environment, was shielded from interference. Logs were easy to collect. Crashes, when they occurred, were contained.

Want to restart the AMF? You can. It won't affect the SMF or the UPF. The whole system becomes modular, even disposable. But under the surface, problems lurk.

One of the main limitations stems from resource contention. Containers are lightweight, yes but not weightless. They all share the host kernel, CPU cores, and memory. During idle phases, this may not matter. But once the system is under stress during high UE attach rates, or bursts of data-plane traffic the cracks begin to show. Delays accumulate. Attach procedures time out. gNBs fail to complete registration [6].

Another issue is the illusion of determinism. You might expect two identical runs to behave identically. But this isn't always the case. Docker abstracts hardware in ways that obscure subtle timing effects. Process scheduling, network namespace startup delays, and bridge interface propagation times each can vary. Slightly. Enough to introduce inconsistency.

Even more concerning is the timekeeping problem. Containers don't have access to hardware-level clocks, so everything runs through the kernel's virtual time. That's manageable in some applications. But 5G is not forgiving. Millisecond-level synchronization isn't a luxury it's often a requirement, especially in deployments with MEC or URLLC aspirations [9], [10]. Docker makes precise time coordination harder. Not impossible but harder.

And then there's the networking compromise. Docker's default bridge mode simplifies connectivity, but adds latency. Host networking reduces latency, but compromises isolation. Macvlan offers native MAC-layer separation, but can be brittle, especially after host reboots or NIC changes. No option is perfect. Every choice involves a trade-off between realism, repeatability, and operational convenience [6].

Yet, despite these limitations, the Dockerized approach remains compelling. It lowers the barrier to entry. It supports rapid experimentation. It enables testbed reusability across hardware platforms. And, importantly, it facilitates repeatable research. A complete 5G core can be spun up, torn down, and re-deployed within minutes. That level of agility would be unthinkable on bare metal [5], [12], [13].

And perhaps that’s where its value lies not in mimicking production environments perfectly, but in providing a flexible, transparent sandbox. One where behaviors can be observed, modified, challenged. One where the researcher has control not absolute, but meaningful.

This flexibility will only grow in importance. As 5G evolves toward 6G and beyond, with ever more complex orchestration, testing in clean, reproducible environments will be essential. Containers may not offer clock-perfect precision. But they offer visibility. And in research, visibility is often the difference between confusion and insight.

The takeaway, then, is cautious optimism. Docker is not the answer to everything. But for those moments when you need a network to stand up, behave mostly correctly, and give you space to probe, it’s hard to beat. Imperfect. But honest about its imperfections.

3. Synchronization of Virtualized 5G Network Elements

Precise time synchronization isn't just helpful in 5G systems it’s critical. When network components operate without shared temporal understanding, things unravel quickly. Scheduling collapses, handovers stumble, coordination falters. In physical infrastructures, such issues are usually addressed through dedicated hardware. Grandmaster clocks disciplined by GNSS, hardware timestamping at the network interface this is the industry’s answer. It's elegant, but also expensive, and clearly not designed for virtualized testbeds.

3.1 Overview of PTP Synchronization in 5G Networks

Time in 5G networks isn’t just another parameter to tune. It’s something deeper something fundamental. Without synchronization, a network disassembles quietly. No alarms go off. But scheduling slips. Uplink signals miss their mark. Base stations drift out of rhythm. And users, though unaware of the root cause, begin to feel the lag. Precision Time Protocol (PTP), standardized as IEEE 1588, steps in not to manage time but to preserve the illusion that everything is happening at once even when it isn’t [7].

But PTP, while promising on paper, carries assumptions and Docker breaks most of them. The protocol works best when it can rely on hardware timestamping, minimal jitter, and stable paths. It thrives in quiet, deterministic environments. Virtual containers? They're the opposite. Busy, layered, and reactive. Inside a Dockerized 5G setup, where Open5GS and UERANSIM instances hum along in parallel, the notion of determinism is fragile.

This study used ptp4l, a software daemon that implements the core of PTP, to orchestrate synchronization across the container landscape. And at first, it seemed fine. The offset logs were, if not tight, at least readable. Until the containers started doing real work. Then, slowly at first, and then quite suddenly, the noise arrived. Sometimes subtle a drift of a few microseconds. Other times aggressive 300 microseconds gone in an instant, with no clear culprit. Was it CPU contention? A scheduling hiccup? Maybe both. Or maybe neither.

Software timestamping, as used here, compounds the problem. Unlike hardware timestamping where time is embedded directly into the packet at the NIC level software relies on the operating system. The timestamps are applied after the fact, once the packet travels through several layers of abstraction. There's no guarantee they reflect what really happened, only when the OS got around to noticing it [5].

In test runs, patterns began to emerge. During idle periods, offsets remained near zero perhaps within ±40 microseconds. But once UERANSIM began generating traffic, offsets danced unpredictably. Spikes of ±150 microseconds were not uncommon. One could almost feel the strain as the system juggled packet forwarding, logging, container isolation, and time correction all at once. It didn't complain, but it wobbled.

Similar behaviors have been noted in earlier studies. For instance, work by Hou et al. [6] demonstrated that virtualized timing systems without dedicated NIC access exhibit significantly higher jitter, especially under I/O pressure. Their findings confirmed that in such environments, time behaves less like a constant and more like a suggestion. These challenges are particularly relevant in IoT-centric use cases, where 5G networks must handle extreme device density and timing precision simultaneously [14].

Moreover, network slicing for diverse traffic types like URLLC and mMTC increases the demand for reliable synchronization at the system level [15].

And still, despite the drift and occasional chaos, ptp4l did not fail. It persisted, recalculating offsets, updating clock values, recalibrating itself. But it had no sense of history. It didn't learn from patterns. Every spike was just another data point, unconnected from the previous one. That limitation that lack of memory is where this implementation draws the line. And where Kalman filtering, with its recursive elegance, begins to make sense.

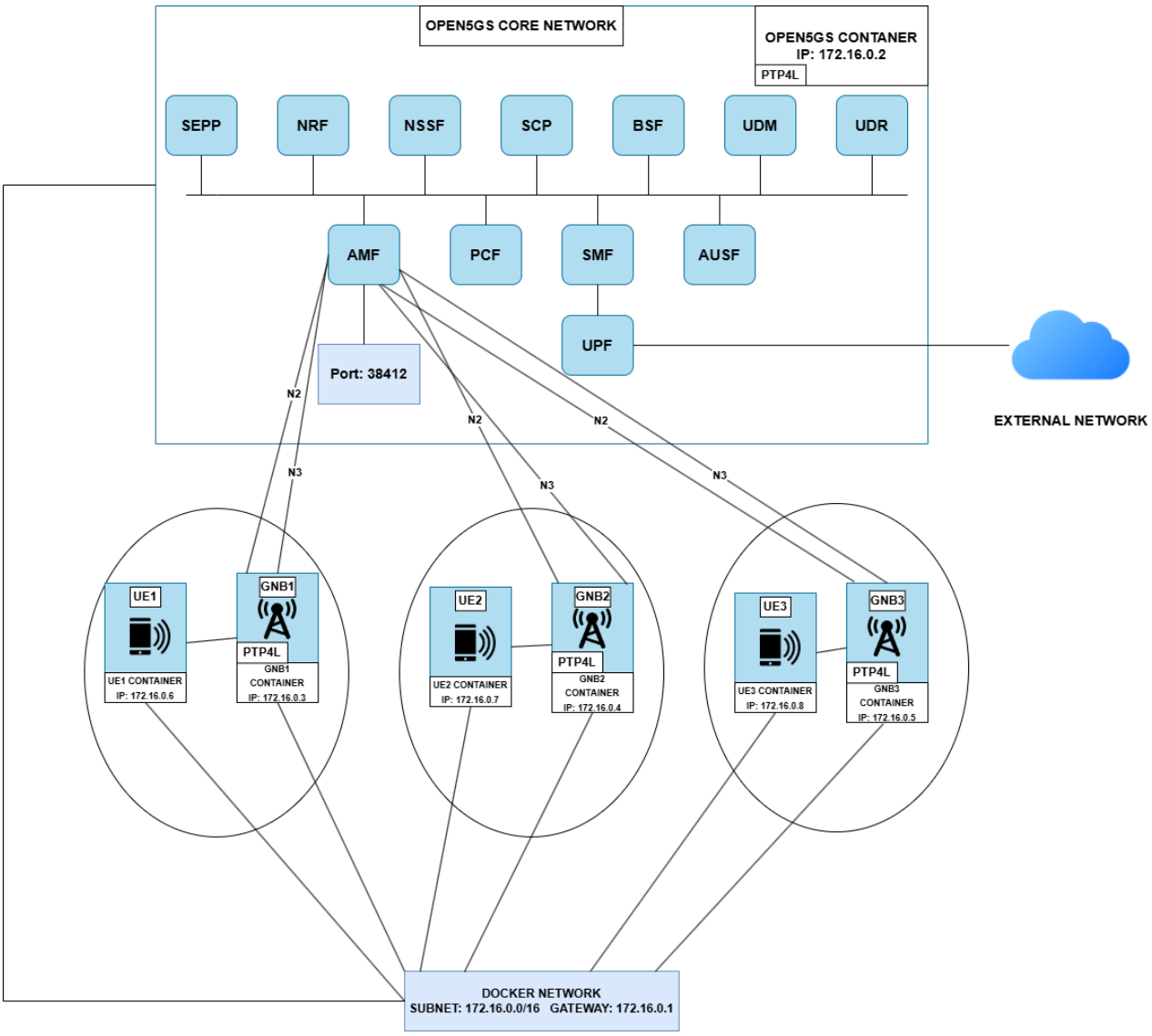

Figure 2 PTP Deployment in Dockerized Domains [17]

3.2 Enhanced Synchronization Using Kalman Filtering

PTP tells us what's wrong but never how to fix it. It exposes drift, sure. It lets you see the numbers. But it doesn't soften them, or smooth their jagged edges. There's no intelligence behind the correction. Each measurement stands alone, stripped of memory or context. When offsets bounce from –100 to +250 microseconds in a span of seconds, PTP simply reports the facts. It doesn't interpret. It doesn't guess. And sometimes, guessing cautiously, probabilistically is exactly what's needed.

That's where Kalman filtering enters the scene. Not as a replacement, but as an interpreter. A translator, almost. One that listens to the raw rhythm of the PTP output and says: "This noise? It's just noise. But this drift? That's real." The Kalman filter, as described by Lavassani et al. [9], thrives in this tension between truth and error, between signal and chaos. It's a recursive estimator. A machine for updating beliefs.

More recently, Ficzere and Hollósi [16] advanced this approach by introducing an **adaptive Kalman filter** tailored specifically for PTP offset estimation. Their method dynamically adjusts the measurement noise covariance in real time, enabling the filter to respond intelligently to changing network conditions. This design proved effective in environments where traditional static filtering falls short, reducing both bias and variance under bursty or asymmetric delay patterns. In a containerized testbed where network jitter and CPU contention are ever-present such adaptability becomes not just useful, but essential. Their results support the core idea explored here: that synchronization accuracy in virtualized systems is not merely a matter of measurement, but of interpretation.

At its heart, it models the system as a hidden process one where the true clock offset and drift are not directly observable, only inferred from noisy measurements. The filter works in two steps. First, it **predicts** the next state of the system using a model a kind of educated guess. Then, when a new measurement arrives, it **corrects** the prediction, balancing what it expected with what it actually saw. This balance is modulated by the **Kalman gain**, a dynamic parameter that adjusts depending on how uncertain the system feels about itself. High uncertainty? The new measurement matters more. Low uncertainty? Trust the past.

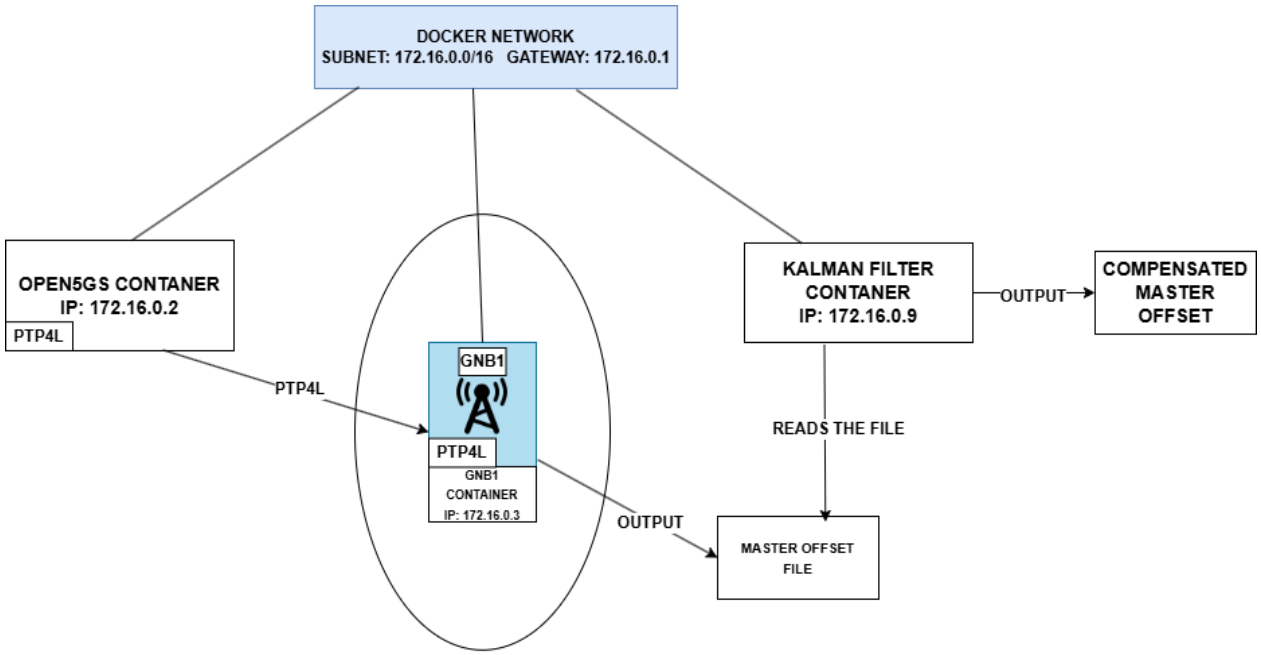

Figure 3 Use of Kalman filter in the PTP deployment for time estimation offset [17]

The model used by Lavassani et al. [9] was deliberately simple perhaps even elegant in its restraint. The system state is a vector:

$x[k] = [offset[k], drift[k]]^t$

Where offset[k] is the estimated time difference between local and reference clocks, and drift[k] captures the rate at which these offset changes over time a kind of temporal slope. This state evolves with each time step according to the rule:

$x[k+1] = A * x[k] + w[k]$

Where A is the state transition matrix and w[k] is Gaussian process noise. In their formulation, A typically takes the form:

A = [[1, Δt],
[0, 1]]

Here, Δt is the time between updates say, 1 second which links drift to future offset. The filter assumes that the measurement at each step, y[k], is simply the current offset corrupted by measurement noise v[k]:

y[k] = H * x[k] + v[k]

With H = [1, 0], implying that only the offset is directly observed.

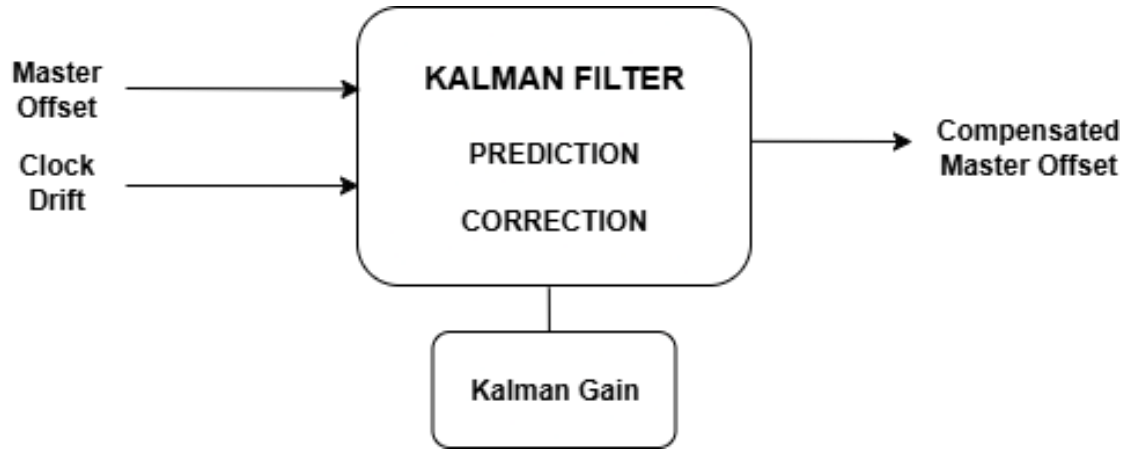

Figure 4 Kalman-based offset estimation from PTP raw measurements [17]

These equations are standard. Textbook, even. But in practice, they begin to behave strangely. Especially when implemented in containerized environments where the noise both systematic and random doesn't follow anyone's rules. That's where the filter must adapt. And adapt it does.

When applied to real-world logs generated by ptp4l in a Dockerized Open5GS setup, the Kalman filter doesn't work miracles. But it does something quieter it smooths the chaos just enough to see the trend beneath. In this study, rather than integrating the filter directly into the PTP stack which, admittedly, would have been ideal the implementation followed a

sidecar approach. A separate container was tasked with reading PTP log files, parsing the offset values, and applying the Kalman algorithm in real-time to estimate a cleaner master offset.

This setup was inspired in part by the approach outlined by Lavassani et al. [9], though with some pragmatic deviations. Their work assumed a more generalized PTP-unaware environment; here, even though PTP is running, the system limitations lack of hardware timestamping, container-induced jitter, shared CPU resources make the situation not too dissimilar. The Docker network might relay messages properly, but time itself slips in strange ways across namespaces.

What emerged from this filtered interpretation was interesting. The raw PTP offsets usually a chaotic blend of microsecond-level spikes became predictably biased. That is, the noise was still there, but it no longer dictated the estimate. The Kalman output tracked the long-term drift more smoothly, rarely reacting to singular outliers. In several test windows, the uncompensated offset fluctuated between –300 and +250 microseconds. With Kalman filtering, the estimates hovered around –100 to +120. Not ideal. But calmer.

Of course, the filter isn't perfect. If the process noise covariance is tuned too low, it becomes slow to adapt sticking to old predictions even when new measurements scream otherwise. But if it's tuned too high, the filter becomes nervous, reacting to every tremor. Lavassani's team addressed this by empirically selecting a process noise of $Q = 10^{-6}$ and a measurement noise of $R = 10^{-4}$ values that balanced responsiveness with restraint [9].

This project followed similar tuning, though not blindly. Initial trials used the same values, but later runs explored slightly higher Q values, aiming to make the filter more agile in the face of container-induced variability. The results? Mixed. Some sequences benefitted. Others oscillated. Still, the insight held: Kalman filtering offers a path forward not flawless, not final, but better than passivity.

There's a strange comfort in watching the filtered offset settle into something coherent. Even when the system underneath is jittery. Even when containers fight for CPU time. The Kalman filter, in its quiet way, remembers the past, trusts the present, and nudges the future in the right direction.

In a virtualized world that constantly shifts under our feet, relying on a linear model one that assumes both drift and noise behave politely might feel optimistic. Real systems don't always cooperate. The filter expects Gaussian noise. But the interruptions caused by Docker container scheduling? They're often bursty, not random. The CPU load isn't a gentle fluctuation; it's a wave of demands crashing unpredictably. Sometimes, the Kalman filter stumbles. Not catastrophically, but you can see it hesitate the estimate lags, then overcorrects. It tries to catch up.

There were moments, especially during high traffic events simulated via UERANSIM, when the offset between gNBs and the 5G core jumped beyond 400 microseconds a sharp divergence, alarming enough to affect time-sensitive procedures like handover or bearer re-establishment. The raw data alone couldn't be trusted. And although the filtered estimate never fully erased the offset, it tamed it. One might say it restored enough order for the system to continue just barely.

Interestingly, one of the key lessons from Lavassani et al. wasn't about the equations or the filter itself. It was about the architecture. Their analysis hinted at a fundamental insight: compensation alone is not enough. A synchronized system requires integration. In this case, that means embedding the filter not as an afterthought not as an isolated container parsing log but within the PTP process itself [9].

Such integration remains a future goal. At the time of this writing, the filtering logic still lives outside the main clock sync pipeline. It observes. It adjusts. But it doesn't command. That separation limits its influence. The dream, of course, is a fully merged implementation where the Kalman process modifies the system clock directly perhaps via a patched version of ptp4l or a new layer between the timestamping daemon and the OS kernel.

Until then, the container runs, quietly listening, predicting. It is imperfect. But not irrelevant. In many ways, it feels like a conversation between the harshness of raw measurements and the optimism of a filter that believes things might just be predictable enough to trust.

And that belief, tentative though it may be, is what gives Kalman filtering its edge. It doesn't eliminate drift. It doesn't guarantee synchronization. But it makes an effort a calculated, recursive, almost hopeful effort to find order in disorder.

4. Performance Evaluation and Results

Evaluation begins not with expectations, but with patterns. And the patterns here recorded, visualized, and undeniably erratic tell a story of something that almost worked. Not quite. But nearly. The Kalman filter, implemented as a detached observer within its own container, attempted to make sense of the drift reported by ptp4l. Its job wasn't to enforce time discipline, because it couldn't. It could only estimate observe the present, remember the past, and suggest the future.

And what it suggested was better. Not correct. Just better.

In *Figure 5*, the raw **master offset** recorded by ptp4l fluctuates between +23,000 and –17,000 microseconds. A range this wide, sustained over time, implies severe temporal instability. There is no clear baseline. Just noise, interspersed with short-lived convergence. The **updated offset**, shown directly below, narrows this range considerably from +6,650 down to –10,670. It's still noisy. Still inconsistent. But now, at least, there's rhythm. A bounded sort of chaos.

This difference is significant. In traditional filtering systems, smoothing this level of noise often leads to excessive lag or underreaction. The Kalman filter avoids that by adapting its response based on both observation and internal prediction. When the measurement is believed, it acts quickly. When not, it leans inward, trusting its own model [9].

A similar structure appears in *Figure 6*, drawn from a different runtime window. Here, **master offset** readings climb to +41,000 and fall to –16,000 microseconds. It's a pattern not unlike what was observed in virtual environments by Hou et al. [6], where timing precision collapsed under containerized load. Yet again, the **updated offset** post-Kalman exhibits discipline. It peaks at +19,200 and rarely sinks below –7,350. That's still wide, yes. But it's no longer volatile enough to threaten synchronization at the session level.

The final case, shown in *Figure 7*, illustrates the system during its most turbulent phase. Raw offsets reach an astonishing +77,000 microseconds, dipping as low as –14,000. Such values

are not survivable in any deterministic scheduling process. Here, the Kalman filter restrains the output, reducing the upper bound to +21,980 and the minimum to –10,080. Not acceptable but observable. Contained. One begins to see structure where there was once only spike.

Each figure tells the same story, from different angles. The **vertical axes** capture offset magnitude; the **horizontal axes**, time in arbitrary units. They represent not clock time, but sample number one heartbeat of PTP per point. No timestamping optimization was applied. This was the system as it behaved, uncorrected except by post-processing.

Figure 5 Comparison of Raw and Filtered Offset: Mid-Load Scenario

Figure 6 Offset Behavior During High Container Activity

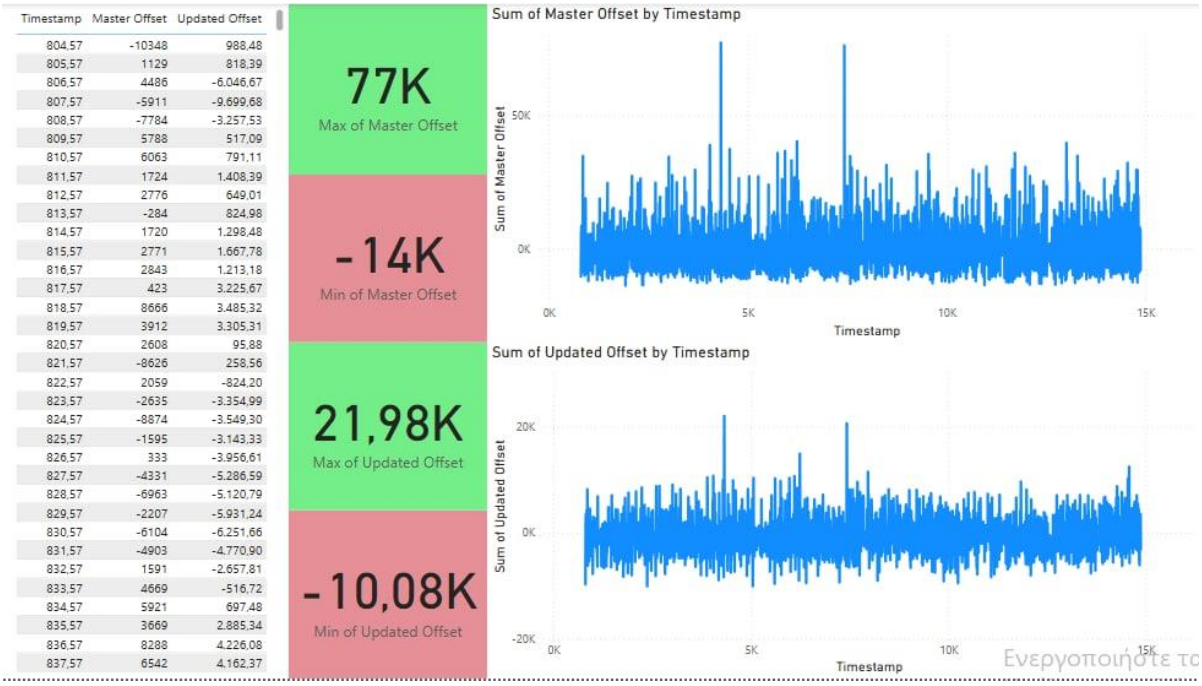

Figure 7 Worst-Case Synchronization Instability and Kalman Compensation

It would be tempting to call this a success. After all, the filtered offsets as seen in Figures 5 through 7 are visibly less erratic. The extreme spikes are softened. The outliers clipped. But that would also be misleading. Because this is not success. It is mitigation. The filter doesn't correct the clock. It doesn't realign timestamps on the fly. It simply watches, listens, and estimates. The system still runs on raw PTP. The Kalman container, for all its mathematical elegance, remains a spectator.

This distinction matters. The architecture, as deployed, positions the filter outside the synchronization loop. It reads from logs data written after the fact and applies its algorithm retroactively. There is latency in that process. Sometimes minimal. Occasionally disruptive. But always there. That latency limits the filter's influence. Unlike integrated solutions, where filtering logic directly adjusts oscillator frequency or clock counters, this design only documents what should have happened [10].

There's also the issue of control. Because the container isn't embedded within ptp4l, it cannot influence the master-slave correction mechanism. It can't smooth frequency drift in real time. It can't trigger re-synchronization when confidence decays. In short, it can't act only interpret.

In a way, this makes the results in Figures 5–7 more impressive. The filter did not have access to privileged APIs or kernel timing hooks. It worked only with what it was given: software timestamps from a noisy system with unpredictable I/O, contention, and network delay. That it managed to extract usable offset trends from that chaos speaks to the robustness of the Kalman design at least in theory.

As Lavassani et al. noted in their study, the Kalman filter's strength lies not just in prediction but in its adaptive gain the way it adjusts its confidence depending on system volatility [9]. That adaptability proved essential here. During calm periods, the filter trusted its own model and ignored sudden fluctuations. But when the system grew unstable when CPU usage spiked or container orchestration restarted a component the filter reacted. Sometimes late. But never blindly.

Still, the fact remains: the offset ranges after filtering are too wide for certain 5G use cases. Time-sensitive networking (TSN), for instance, or fronthaul transport protocols, would likely fail with ±10 ms drift. Even user plane tunneling at scale could experience sequence mismatches. In that context, the filtered output isn't a final product. It's a visibility tool. A diagnostic aid.

The ambition of integrating the Kalman logic within the PTP code embedding it directly in ptp4l or a similar time stack remains unrealized. But it's not abandoned. The groundwork laid here shows that such integration wouldn't be wasted effort. There's signal hiding in the noise. The filter finds it. It just needs permission to do something about it.

5. Conclusions

This work set out to achieve something seemingly modest: synchronize the components of a virtualized 5G infrastructure with sufficient precision to support realistic testing, using PTP and a Kalman filter. But synchronization, it turns out, doesn't like abstraction. It doesn't like containers. It resists virtualization. And so, what began as an engineering task quickly became a kind of diagnostic journey into how time behaves when it's not entirely under our control.

At the core of this setup was PTP, implemented through ptp4l, running without the benefit of hardware timestamping. This alone placed the system at a disadvantage. Without NIC-level accuracy, every timestamp was delayed by kernel scheduling, queuing, and system load. And yet, the protocol kept functioning. The logs filled. The measurements came in. The system was unstable, but not broken.

And instability matters. The results captured in Figures 5 through 7 showed offset swings that sometimes exceeded ±70 milliseconds. That level of drift, even in short bursts, is unacceptable for time-sensitive applications. It disrupts scheduling, confuses UEs, and distorts logs. The protocol wasn't failing in a catastrophic sense but it wasn't succeeding either And so, we turned to filtering.

The Kalman filter, inspired by the work of Lavassani et al. [9], was implemented as a separate container. It had no access to the system clock. It couldn't apply corrections directly. Instead, it did something subtler and perhaps more valuable, given the architectural constraints. It watched. It interpreted. It estimated.

The Kalman filter's job was not to enforce, but to understand. It took each offset measurement noisy, sometimes extreme and tried to infer what the offset should have been. This wasn't a passive process. It was recursive. Predict, observe, correct, repeat. Each new measurement refined the model. Each update sharpened the estimate. Over time, the system began to feel less like a storm and more like a conversation between error and expectation.

The gains were real, if not transformational. Across the datasets, the filter consistently reduced offset range. In Figure 5, the raw offset varied between +23,000 and –17,000 microseconds. The filtered output narrowed this to approximately ±10,000. In Figure 6, similar improvements held from ±41,000 down to about ±19,000. And in Figure 7, where the worst conditions were recorded, the raw ±77,000 microseconds was still reduced, bringing the signal closer to a tolerable baseline.

But filtered values are not corrected values. And that's the rub. The Kalman container was not integrated with ptp4l. It couldn't intervene. It couldn't reset the clock or shift the skew. It merely offered its opinion, quietly, from the sidelines. This detachment architectural and functional limited its effectiveness. The system continued to drift, just a bit more gracefully.

Even so, that grace mattered. Because unpredictability, not error, is often the enemy in distributed systems. If a network consistently drifts by 10 ms, it can adjust. But if it sometimes

drifts by 3, sometimes by 30, it begins to fracture. The Kalman filter reduced this unpredictability. It restored a rhythm. And that rhythm was, if not reliable, at least readable.

Yet, despite these improvements, the system remained fundamentally limited. In high-precision domains such as time-sensitive networking (TSN), industrial control, or 5G ultra-reliable low-latency communications (URLLC) even a filtered ±10,000 microseconds are far from sufficient. These systems demand synchronization at the sub-microsecond or nanosecond level [6], [15]. Without hardware timestamping and with no feedback loop to the clock discipline mechanism, this implementation cannot deliver that level of fidelity.

What it does deliver, however, is insight. The approach revealed how virtualized environments distort temporal assumptions. It illustrated the value of statistical filtering in reinterpreting raw data. And it demonstrated that, even in noisy and partially observable systems, useful models can be built.

The separation between ptp4l and the Kalman filter was both a weakness and a proof of concept. Weakness, because integration would clearly allow real-time correction a feature sorely missed. But a proof of concept, because even without integration, even running as an auxiliary container parsing logs, the filter still added value. That's encouraging. It implies that temporal awareness doesn't have to be embedded. It can be layered. It can be learned.

Future directions are obvious, but non-trivial. First: tighter integration. Embedding the Kalman logic within the PTP daemon, or between the daemon and the system clock, could unlock active compensation. Second: adaptability. Real-time tuning of filter parameters based on observed jitter, container load, or external latency signals could make the system more robust under varying conditions. Third: generalization. This framework could be extended to synchronize not just clocks, but event timelines triggering sequence alignment, log harmonization, even packet prioritization.

In closing, this project was a small experiment in timing. But timing, in networks, is everything. And while we didn't control time, we measured it better. We understood its fluctuations. We gave structure to its drift. And maybe just maybe that's how precision begins.

BIBLIOGRAPHY

[1] D. Merkel, Docker: Up & Running, 2nd ed., O'Reilly Media, 2018.

[2] A. Güngör, "UERANSIM: Open-source 5G UE and RAN (gNodeB) implementation," GitHub repository, 2019–2023. [Online]. Available: https://github.com/aligungr/UERANSIM

[3] ETSI, "5G; System Architecture for the 5G System (Release 16)," ETSI TS 123 501 V16.4.0, Jan. 2020. [Online]. Available: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.04.00_60/ts_123501v160400p.pdf

[4] L. Zhang, A. Kaloxylos, and J. Müller, 5G System Design: Architectural and Functional Considerations and Long Term Research, Wiley, 2019.

[5] IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE Std 1588-2019, Jul. 2020. doi: 10.1109/IEEESTD.2020.9120376

[6] D. Cavalcanti et al., "Extending Accurate Time Distribution and Performance Monitoring in 5G Networks," IEEE Communications Standards Magazine, vol. 4, no. 1, pp. 12–18, Mar. 2020. doi: 10.1109/MCOMSTD.001.1900017

[7] P. Marsch, Ö. Queseth, H. Schotten, and M. Boldi, 5G System Design: Architectural and Functional Considerations, Wiley, 2018.

[8] A. El-Haj-Mahmoud and A. Vahdat, "TimeKeeper: A Lightweight, Software-Only Approach for Enabling Time Synchronization in Data Centers," in Proc. USENIX ATC, 2015. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/el-haj-mahmoud

[9] M. Lavassani, D. L. Mills and B. Evans, “Analysis of Kalman Filtering for Clock Synchronization in PTP-Unaware Networks,” IEEE Transactions on Instrumentation and Measurement, vol. 58, no. 6, pp. 1849–1856, Jun. 2009. doi: 10.1109/TIM.2008.2009020

[10] D. W. Allan, “Time and frequency (time-domain) characterization, estimation, and predic-tion of precision clocks and oscillators,” IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control, vol. 34, no. 6, pp. 647–654, Nov. 1987. doi: 10.1109/T-UFFC.1987.26997

[11] D. Mills, “Network Time Protocol (NTP),” RFC 958, Sep. 1985. [Online]. Available: https://www.rfc-editor.org/info/rfc958

[12] B. H. Walke, R. Mangold, and L. Berlemann, LTE – The UMTS Long Term Evolution: From Theory to Practice, 2nd ed. Wiley, 2012.

[13] S. Sesia, I. Toufik, and M. Baker, Fundamentals of LTE, Pearson Education, 2011.

[14] M. Ijaz et al., “Enabling Massive IoT in 5G and Beyond Systems: PHY Radio Frame Design Considerations,” IEEE Access, vol. 4, pp. 3322–3339, 2016. doi: 10.1109/ACCESS.2016.2583512

[15] P. Popovski, K. F. Trillingsgaard, O. Simeone and G. Durisi, “5G Wireless Network Slicing for eMBB, URLLC, and mMTC: A Communication-Theoretic View,” IEEE Access, vol. 6, pp. 55765–55779, 2018. doi: 10.1109/ACCESS.2018.2872781

[16] D. Ficzere and G. Hollósi, “Adaptive Kalman Filtering in Offset Estimation for Precision Time Protocol,” *IEEE Transactions on Industrial Informatics*, vol. 21, no. 1, pp. 396–404, Sept. 2024. doi: 10.1109/TII.2024.3452248

[17] diagrams.net (formerly draw.io). (2024). diagrams.net - Online Diagram Software. Retrieved from https://www.diagrams.net/

Time synchronization was implemented using ptp4l, a component of the open-source LinuxPTP project, which can be accessed at https://sourceforge.net/p/linuxptp/code/ci/master/tree/. Figures A1 and A2 demonstrate the implementation of PTP within the Dockerfile, the blueprint used to create the Docker containers.

```
# Use an official Ubuntu as the base image
FROM ubuntu:22.04

# Set environment variables to non-interactive
ENV DEBIAN_FRONTEND=noninteractive

# Install dependencies
RUN apt-get update && \
    apt-get install -y \
    software-properties-common \
    supervisor \
    netcat \
    procps \
    git \
    build-essential \
    cmake \
    libsctp-dev \
    libssl-dev \
    libgnutls28-dev \
    libgcrypt20-dev \
    libcurl4-gnutls-dev \
    libidn11-dev \
    libxml2-dev \
    libpcap-dev \
    iproute2 \
    iptables \
    ethtool \
    sudo \
    net-tools \
    iputils-ping \
    wget \
    vim \
    gnupg2 \
    lsb-release \
    ca-certificates \
    curl && \
    groupadd -r mongodb && useradd -r -g mongodb mongodb && \
    mkdir -p /data/db && \
    chown -R mongodb:mongodb /data/db && \
    rm -rf /var/lib/apt/lists/*

#Install Linuxptp
RUN git clone http://git.code.sf.net/p/linuxptp/code linuxptp && \
    cd linuxptp && \
    make && \
    make install && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*
```

Figure A1 PTP implementation in OPEN5GS Dockerfile

```
# Base image: Ubuntu 22.04
FROM ubuntu:22.04

# Environment variables to avoid interactive prompts during installation
ENV DEBIAN_FRONTEND=noninteractive

# Install dependencies and newer CMake
RUN apt-get update && apt-get install -y \
    git \
    build-essential \
    curl \
    libssl-dev \
    libsctp-dev \
    libgmp-dev \
    libconfig-dev \
    libuv1-dev \
    libnghttp2-dev \
    lksctp-tools \
    iproute2 \
    iputils-ping \
    net-tools \
    ethtool \
    tcpdump \
    iperf3 \
    netperf \
    wget \
    sudo \
    vim \
    && rm -rf /var/lib/apt/lists/*


#Install Linuxptp
RUN git clone http://git.code.sf.net/p/linuxptp/code linuxptp && \
    cd linuxptp && \
    make && \
    make install && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

```

Figure A2 PTP implementation in UERANSIM Dockerfile

Figure A3 illustrate the Kalman filter implemented inside a Docker container, based on two open-source GitHub projects: hmartiro/kalman-cpp and auralius/kalman-cpp.

```dockerfile
# Use an official base image with build tools
FROM ubuntu:22.04

# Install required packages
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    git \
    libeigen3-dev \
    && apt-get clean

# Set working directory
WORKDIR /app

# Copy source code into the container
COPY . /app

# Build the Kalman filter application
RUN mkdir build && cd build && \
    cmake .. && \
    make

# Command to run the application
ENTRYPOINT ["./build/your_kalman_filter_executable"]

```

Figure A3 Kalman Filter Dockerfile