Quartz: *Time-as-a-Service* for Coordination in Geo-Distributed Systems

Sandeep D'souza
Carnegie Mellon University

Heiko Koehler
Nutanix Inc.

Akhilesh Joshi
Nutanix Inc.

Satyam Vaghani
Nutanix Inc.

Ragunathan (Raj) Rajkumar
Carnegie Mellon University

Abstract

Geo-distributed systems ranging from databases to *cyber-physical* applications increasingly rely on a shared and precise notion of time to achieve coordination. This is especially true for cyber-physical applications ranging from local-scale robotic-coordination and city-scale traffic management to regional/planetary-scale smart grids. Each of these applications utilizes event orderings and timing offsets to make *real-time* decisions, so as to perform coordinated *action* at their distributed endpoints. The emergence of edge computing, specifically to facilitate low-latency decision-making, is leveraging the trend where multiple cyber-physical and software applications with different *timing* requirements will coexist in both the cloud and at the edge. To enable such fault-tolerant *time-based* coordinated applications running on multi-tenant geo-scale infrastructure, we introduce the Quartz framework, which exposes *Time-as-a-Service*. Quartz allows geo-distributed application components to each specify its timing requirements, while it *autonomously* orchestrates the underlying infrastructure to meet them. Centered around a shared *virtualized* notion of time, based on the *timeline* abstraction [1], Quartz provides an API which makes it easy to develop time-based geo-distributed applications. Using this API, Quartz feeds back the timing uncertainty, i.e., the delivered *Quality of Time (QoT)* [1] back to each application, enabling it to be fault-tolerant in the face of clock-synchronization failure. Quartz is designed for containerized applications, features a distributed architecture and is implemented using containerized micro-services. Experimental evaluations on real-world embedded, edge and cloud platforms highlight the performance and scalability of our architecture.

SEC '19, November 07–09, 2019, Washington DC

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363311

CCS Concepts

• **Computer systems organization → Embedded and cyber-physical systems**.

Keywords

time-as-a-service, cyber-physical systems

ACM Reference Format:
Sandeep D'souza, Heiko Koehler, Akhilesh Joshi, Satyam Vaghani, and Ragunathan (Raj) Rajkumar. 2019. Quartz: *Time-as-a-Service* for Coordination in Geo-Distributed Systems. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318216.3363311

1 Introduction

Coordination is key to the successful operation of a distributed system. Distributed coordination occurs at different spatial and temporal scales, ranging from local-scale robotic coordination – occurring at the timescale of hundred microseconds to a few milliseconds, to planetary-scale coordination among GPS satellites – occurring even at the nanosecond timescale. A non-exhaustive list of such coordinated systems includes swarm robotics [2], distributed databases [3], tele-surgery [4], industrial robotics [5], smart grids [6] and connected vehicles [7]. Figure 1 highlights the spatio-temporal nature of distributed coordination.

The common thread binding many of the above-mentioned applications is the need for low-latency decision-making. This is especially true for *cyber-physical* applications, which involve the cyber components of computation and networking, interacting with the physical world [8]. In these systems, the nature of coordination is usually dependent on the analysis of *sensed* data by intelligent *computational* entities, which in real-time decide a course of coordinated *action/actuation* at distributed endpoints. The data-intensive and low-latency nature of decision-making makes the cloud in tandem with edge cloudlets well-suited for hosting such applications. While recent work [9–12] has focused on using edge computing to reduce latency, the need for a distributed coordination primitive has not received much attention.

Time is one such construct which plays an important role in enabling coordination among distributed entities [13]. This is especially true for cyber-physical systems (CPS) which need to interact with the real world. A shared notion

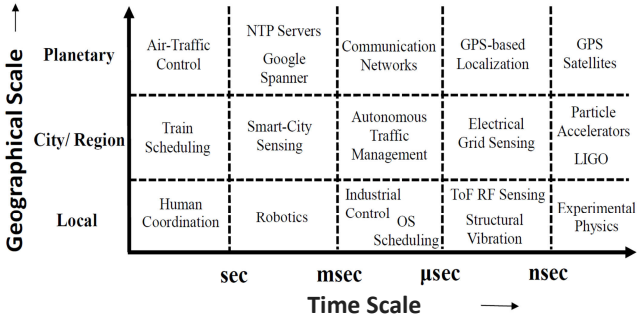


Figure 1: The scale of coordination in time and space

of time, by means of synchronized clocks, enables: (i) events to be ordered at distributed scale, and (ii) coordinated actuation to be scheduled at/by specific time instants. Therefore, maintaining a shared notion of time is critical to the reliable operation of many large-scale distributed systems.

Clock synchronization is a mature field and technologies such as GPS, Network Time Protocol (NTP) [15], and Precision Time Protocol (PTP) [16] have made it possible to provide distributed systems with a reliable and accurate shared notion of time. However, these technologies are best-effort and/or agnostic to application-specific requirements. Additionally, clock synchronization is not perfect, and there is always some uncertainty in a node's estimate of the shared notion of time. This timing uncertainty is introduced by a variety of factors including, but not limited to, networking delays [15], timestamping errors, and operating system and virtualization-induced latency and jitter [17][18]. If this timing uncertainty exceeds an application's specifications, it can affect the quality and reliability of coordination [19]. The level of uncertainty acceptable to an application often depends on the time granularity at which coordination occurs, as well as the coordination policy [19].

As time is fundamental to a range of applications, it needs to be exposed *as a service* to applications [14]. Therefore, we define *Time-as-a-Service* (TaaS) as "the ability to provide an application-specific clock, which tracks a time reference, such that the timing uncertainty does not exceed application-specified requirements."

Time exposed as a first-class entity to applications is key for TaaS. This can be done by: (i) allowing applications to specify their timing requirements in terms of accuracy and resolution, and (ii) feeding back the delivered timing uncertainty back to the application. The latter enables applications to adapt if timing uncertainty exceeds specified limits. Thus, fault-tolerant time-based coordination becomes enabled by using the notion of *Quality of Time* (QoT) [1], which represents the *end-to-end* uncertainty bounds corresponding to a timestamp, with respect to a clock reference. From an application perspective, if these bounds exceed an acceptable limit, the application can enter a graceful degradation mode, and thus be fault-tolerant during clock-synchronization failure.

Based on the notion of QoT, [1] also introduced a reference *QoT Architecture* along with its corresponding LAN-scale implementation, called the *QoT Stack for Linux*. The QoT Stack features a kernel-module-based implementation and introduces a preliminary prototype to demonstrate the benefits of exposing time as a first-class entity to applications. However, its use of kernel-space components restricts scalability and introduces portability issues, which limit its utility for deployments on public infrastructure.

Modern distributed applications are inherently complex, and consist of multiple interacting components. Thus, deploying these components and managing their life-cycles are complex endeavors. Additionally, many of these components will be deployed in the cloud or at the edge in conjunction with other applications. In such scenarios, the use of OS-level virtualization technologies like containerization [20] simplifies the deployment and life-cycle management of distributed applications. Thus, Quartz builds on the QoT Architecture [1] for providing *Time-as-a-Service* (TaaS) to containerized applications. Quartz features a distributed modular architecture, and is implemented using containerized micro-services, making it easy to deploy and use across a range of platforms.

Unlike the kernel-space QoT Stack [1] which operated at LAN-scale, Quartz overcomes the scalability and portability issues by featuring a fully user-space implementation which (i) supports multi-tenancy, (ii) operates at geo-distributed (WAN)-scale, and (iii) is portable to an array of application domains and platforms. Quartz also provides an API for distributed coordination based on the *timeline* abstraction [1], and allows distributed application components to specify their required QoT. Based on these requirements, Quartz orchestrates the underlying system and clock-synchronization protocols to meet these application-specific requirements, and feeds back the delivered QoT back to the application.

The key contributions of this paper are as follows:

(1) Elucidating the challenges and architectural choices in exposing Time-as-a-Service (TaaS), maintaining timelines and estimating QoT at geo-distributed scale.
(2) Introducing techniques to make clock-synchronization protocols, adaptive to application QoT requirements.
(3) Introducing Quartz, an autonomous, adaptive and fault-tolerant middleware exposing TaaS for containerized applications using time as a coordination primitive.

2 An Application's Perception of Time

We first motivate the utility of Quartz by describing two application scenarios which can be enabled by using a shared notion of time and QoT. These applications are (i) *DronePorter*, a fleet of drones coordinating to transport a payload, and

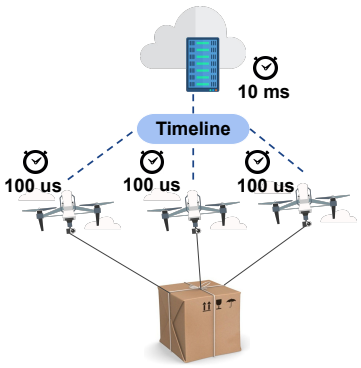


Figure 2: *DronePorter*: Drone coordination

(ii) *TimeCop*, a traffic-management solution which coordinates vehicular traffic flow at city scale in both space and time. However, the core concepts can be adapted to other distributed-coordination application domains. We start by describing the timeline abstraction introduced in [1].

Timelines: In [1], the authors introduced the *timeline* abstraction, which abstracts away clock synchronization from applications. A timeline provides a shared *virtual* clock reference to all the distributed components of an application. Consider an application that needs to perform coordinated actions at its distributed endpoints. All these components bind to a common timeline, each specifying its QoT requirements. As a result, the timeline abstraction provides the following functionalities: (i) allows an application to specify which components coordinate with each other using *shared time*, and (ii) provides visibility into where each application component is deployed, and what its QoT requirements are with respect to the timeline reference. This allows the underlying framework to orchestrate the clock-synchronization protocols to ensure that QoT requirements are met, while making the achieved QoT visible to the application.

A timeline is not necessarily tied to any standard timing reference (such as UTC), and, in the context of distributed coordination, serves as a "narrow waist" [18]. This enables developers to easily develop distributed time-based applications on heterogeneous infrastructure, using a timeline-based API. The ability of a timeline to expose a virtual clock reference allows different coordinating sub-groups with varying QoT requirements to each have its own time reference and coexist on the same infrastructure [1]. Note that each node bound to a timeline can have different QoT requirements with respect to the chosen reference. These QoT requirements are generally defined by (i) safety constraints, (ii) performance requirements and/or (iii) the assumptions/tolerances of the controller/decision-making entity. Additionally, multiple *virtual* timelines can coexist on a single node.

DronePorter: Consider a fleet of n drones (as shown in Figure 2) transporting an object Ω, too large to be carried by a single drone. To successfully transport Ω, the drones need to follow a coordinated flight-plan such that (i) the object is not damaged or destabilized, and (ii) the drones do not collide with each other or obstacles in the environment. One way to accomplish this is by having a master entity, which can be one of the drones, send out timestamped flight-plans with way-points to each of the drones, such that each drone tries to reach a given way-point at the specified time.

To coordinate successfully, the clock on each drone needs to be synchronized such that the accuracy is within some specified limits. This accuracy (or uncertainty) specification can depend on multiple factors, ranging from the velocity and size of the drones, to the other uncertainties in the environment. For example, to meet a particular velocity, while maintaining safety, having a tighter clock-synchronization accuracy can be used to compensate for higher localization uncertainties or higher environmental uncertainties [21]. Therefore, in this scenario, each drone can use Quartz to bind to a timeline each specifying its QoT requirements. If the QoT deviates beyond these requirements, the drones can be notified, and can adapt by moving into a graceful-degradation mode. Additionally, as shown in Figure 2, we can also have an edge/cloud controller also join the timeline, and provide (i) high-level objectives/guidance to the fleet of coordinating drones, and (ii) fleet-management capabilities.

TimeCop: Consider a city with an adaptive traffic signal deployed at each intersection, which contains: (i) a traffic signal with an interface through which the phase (traffic-signal state) can be set, and (ii) camera-based sensors which provide per-lane queue lengths (number of vehicles).

Each intersection is controlled by a traffic controller, which can be deployed on an edge device at or near the intersection, for low-latency decision-making. This controller is responsible for controlling the timing and phase of the traffic signals at the intersection. The traffic controller makes decisions periodically, by taking as input (i) the number of vehicles per ingress lane at the intersection (read from the traffic sensors) in the last interval, and (ii) the number of vehicles inbound from adjacent intersections (published by adjacent intersections). The generated output is the next phase of the traffic signal. In this scenario, a shared notion of time is key to ensure that (i) the state from adjacent intersections has accurate timestamps, and (ii) the phase of the traffic signals at an intersection can be switched at an accurate time instant to ensure efficient traffic flow. Thus, each intersection controller uses Quartz to bind to the *traffic-management* timeline with a QoT requirement of +/-1 ms, while Quartz ensures that all controllers bound to the timeline share the same notion of time with the desired QoT specification. Thus, the timeline abstraction allows a coordinating group of endpoints to be specified. Quartz also appends every timestamp with accurate QoT estimates, enabling controllers to decide "data validity" based on the QoT bounds, i.e., data with QoT

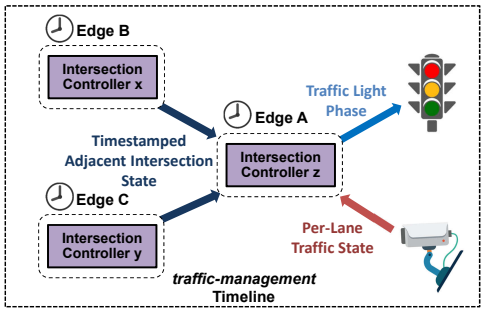


Figure 3: *TimeCop*: City-Scale Traffic Management

bounds beyond tolerable limits can be discarded or used with abundant caution. Figure 3 illustrates the TimeCop solution.

Consider a scenario where multiple applications such as *TimeCop* and *DronePorter* are deployed on the same infrastructure. For example, DronePorter's high-level controller can be deployed on the same edge device as TimeCop's per-intersection traffic controllers. One can also envision a situation where multiple such emerging smart-city applications are deployed on the same infrastructure. In such a scenario, the ability to simultaneously maintain multiple per-application timelines allows (i) each application's coordinating components and their QoT requirements to be individually specified, and (ii) the system to meet the potentially different QoT requirements of each application.

With each application component specifying the required QoT, the system knows the maximum level of uncertainty tolerable by the distributed-coordination application. Since each node independently computes its QoT with respect to the reference, a node can enter a graceful-degradation [27] mode when the level of uncertainty exceeds the tolerable limit. Additionally, if a coordination message is delayed or arrives too late, all a node needs to do is compare the message timestamp against the current time on its local clock [13]. Given that commodity oscillators drift slowly, the probability of clock-synchronization failure is much lower than the probability of CPUs, networks or disks failing [3]. Therefore, utilizing a shared notion of time with the added notion of QoT can enable scalable and fault-tolerant coordination [19].

3 Quartz: *Time-as-a-Service* (TaaS)

We now introduce Quartz which exposes *Time-as-a-Service* to containerized applications. We describe Quartz by starting at the application level and explaining the capabilities Quartz provides through its API. Subsequently, we focus on its architecture and its implementation as micro services.

Quartz API: Quartz features a rich API that is centered around the notion of a *timeline* – a virtual sense of time to which applications bind with their desired accuracy level and minimum clock resolution [1]. A timeline is the key primitive specifying the application components which coordinate with each other. The Quartz API provides applications the ability to (i) bind/unbind from a timeline, (ii) specify/update their QoT requirements, (iii) schedule computation, sensing and actuation by/at a reference time instant, (iv) timestamp events and (v) get latency estimates between a pair of nodes on a timeline. For an application involving distributed coordination, latency estimates give a good idea of how far into the future actuation commands should be scheduled.

All API calls return the QoT delivered to the application, providing the ability to adapt to changes in QoT. Table 1 provides an overview of the key API calls supported by Quartz.

Listing 1 shows a simple application written using Quartz's Python API binding. The sample application binds to a timeline with an accuracy and resolution requirement of 1ms each. The application then periodically wakes up every second and reads the time. This is indicative of a collection of periodic time-triggered application components which each wake up at their own specific time instants to perform some coordinated action. Similarly, we can also envision event-driven applications which, in response to an event, capture a timestamp of the event. Such event timestamps can be captured using a callback function facilitated by the **`timeline_timestamp_events`** API call.

Listing 1: Simple Periodic App using the Quartz API

```
def main_func(timeline_uuid: str, app_name: str):
# Initialize the TimelineBinding class as an app
binding = TimelineBinding("app")
# Bind to the timeline with 1ms accuracy and resolution
ret = binding.timeline_bind(timeline_uuid, app_name, 1ms, 1ms)
if ret != ReturnTypes.QOT_RETURN_TYPE_OK:
print ('Unable to bind to timeline, terminating ....')
exit (1)
# Set the Scheduling Period and Offset (1s and 0ns repectively)
binding.timeline_set_schedparams(1000000000, 0)
while running:
# Wait until the next period
binding.timeline_waituntil_nextperiod()
# Do Something -> Read the time with the uncertainty
tl_time = binding.timeline_gettime()
print('Timeline time is       %f' % tl_time["time_estimate"])
print('Upper Uncertainty is   %f' % tl_time["interval_above"])
print('Lower Uncertainty is   %f' % tl_time["interval_below"])
# Unbind from the timeline
binding.timeline_unbind()
```

3.1 Quartz: Architecture & Implementation

To enable time-based geo-distributed applications at scale and deliver Time-as-a-Service, Quartz is tasked with the following primary objectives: (i) maintaining the notion of a timeline at geo-distributed scale, (ii) meeting application-specific QoT requirements with respect to the chosen timeline reference, and (iii) computing QoT estimates with respect to the chosen timeline reference. While meeting the above objectives, Quartz is also tasked with optimizing system resources by *merging* multiple timelines under the hood, based on application requirements and how they are deployed.

Given the above objectives, Quartz needs to overcome the following challenges (i) ***scalability***: both geographical and

Table 1: Quartz API Calls

Category	API Call	Return Type	Functionality
Timeline Association	**`timeline_bind`** (node_name, accuracy, resolution)	timeline	Bind to a timeline
	`timeline_unbind` (timeline)	status	Unbind from a timeline
	`timeline_setaccuracy` (timeline, accuracy)	status	Set Binding accuracy
	`timeline_setresolution` (timeline, resolution)	status	Set Binding resolution
Time Management	**`timeline_gettime`** (timeline)	timestamp+QoT	Get timeline reference time with uncertainty
	`timeline_translate` (timestamp, src_timeline, dst_timeline)	timestamp+QoT	Translate a timestamp on one timeline into another
Event Scheduling & Timestamping	**`timeline_waituntil`** (timeline, absolute_time)	timestamp+QoT	Absolute blocking wait
	`timeline_sleep` (timeline, interval)	timestamp+QoT	Relative blocking wait
	`timeline_set_schedparams` (timeline, period, start_offset)	status	Set period and start offset
	`timeline_waituntil_nextperiod` (timeline)	timestamp+QoT	Absolute blocking wait until next period
	`timeline_timer_create` (timeline, period, start_offset, count, callback)	timer	Register a periodic callback
	`timeline_timestamp_events` (timeline, event_type, event_config, enable, callback)	status	Configure events/external timestamping on a pin
Latency	**`timeline_reqlatency`** (timeline, src_node, dst_node, num_measure, percentile)	duration	Get the latency between two nodes on a timeline

quantitative, (ii) ***autonomy***: the system should adapt to application demands and faults, (iii) ***portability***: easy to deploy and manage, and (iv) ***ease of development***. Challenges (i) and (ii) are influenced by the architecture, while (iii) depends on the implementation, and (iv) depends on the API.

A hierarchical architecture yields both scalability and autonomy. Therefore, Quartz features a 3-tier hierarchical architecture with services which operate at the following tiers:

1) A **Node** represents any single computing node/device (virtual or physical) with an independent clock.

2) A **Cluster** represents any administrator-defined set of networked nodes which can communicate with each other. An example cluster is a set of nodes connected over a LAN. Note that a node cannot belong to more than one cluster, since each node has a single *independent* clock.

3) The **Global** scope represents the global set of clusters.

Based on the scope at which a timeline is discoverable by other nodes, we define two types of timelines:

1) A **Local Timeline** is discoverable only on nodes inside the cluster in which the timeline is created. It is useful for applications with coordination restricted to the cluster scope.

2) A **Global Timeline** can be discovered by any node in the global set of clusters. It is useful for applications which have coordinating components spanning multiple clusters.

When a timeline is created, its type must be specified. This allows Quartz to choose an appropriate clock-synchronization protocol and virtual timeline reference.

We implement Quartz using user-space micro-services, which are designed to run natively or as Docker [20] containers. Each service exposes an interface for exchanging information and receiving requests. Figure 4 illustrates the Quartz Architecture, and highlights the interactions between the various components through their exposed interfaces. We first describe each service's high-level implementation before stating how they provide different functions:

1) The **Timeline Service** is the interface through which applications interact with Quartz, i.e., most API requests are handled by the timeline service. It exposes a unix-domain socket (UDS)-based interface through which applications on the node can send requests to the service. It is also tasked with performing the bookkeeping of the timelines that exist on a node, the applications bound to each timeline, and the QoT requirements of each application and timeline. Therefore, the timeline service maintains timelines at the scope of a node, and hence, each node has its own timeline service.

2) The **QoT Clock-Synchronization Service** synchronizes the per-timeline clocks and computes the QoT estimates. Since every node has a hardware clock, which serves as a basis for per-timeline virtual clocks, each node has its own clock-synchronization service. Like the timeline service, it also exposes a UDS-based interface through which the timeline service can send it requests. In its current implementation, the synchronization service supports NTP [15], PTP [16] and Huygens [26] clock-synchronization protocols.

3) The **Coordination Service** is responsible for maintaining timelines within the scope of a cluster. Hence, every cluster must have one active coordination service. Within a cluster, the coordination service helps each node discover other nodes on a timeline, and conveys QoT requirements across nodes. This information is used by each node's timeline service to orchestrate its node's clock-synchronization service, based on application QoT requirements. It exposes a REST API accessible to all the nodes within the cluster. The REST API allows the timeline service on each node to register (POST) timelines and its QoT requirement with the coordination service. This also allows timeline services on other nodes in the cluster to discover timelines (GET) and update (PUT) the most-stringent QoT requirement on a timeline.

4) The **Global Discovery Service** serves as Quartz's global book-keeper, and is tasked with maintaining timelines at the global scope, by allowing a cluster to *discover* the presence of other timelines and clusters bound to it. The discovery service maintains a key-value store of timelines and their relevant metadata along with the clusters associated with each timeline. It also provides an interface for cluster-specific coordination services to discover each other, and exchange timeline and QoT information. It is implemented using Apache Zookeeper [34], which provides a consistent and highly-available filesystem-like abstraction. The discovery service

maintains a */timelines* Zookeeper node, under which different timelines are registered. This allows cluster-specific coordination services to register the presence of timelines associated with their cluster, as */timelines/<timeline-name>*. Under this timeline-specific Zookeeper node, a child node exists for each cluster participating in the timeline. In particular, the ability to (1) set *watches* on Zookeeper nodes: receive asynchronous notification on changes to a node or its children, and (2) *ephemeral* nodes: elements which disappear on a network disconnect, allows the coordination service to detect if another cluster has joined or left a timeline.

As may be expected, using a hierarchical architecture provides a very clear distribution of responsibilities. Therefore, even if higher-layer services (global or cluster-level) are temporarily lost, lower-layer services (cluster or node-level) can still continue to operate and provide essential functionality.

Quartz Clocks: Quartz also features timeline-specific *clocks*, which are required for providing applications with their own shared notion of time. At the node scope, Quartz utilizes a core clock C_{core} [1] derived from a hardware clock, which maintains a monotonic free-running notion of time with undisciplined drift and offset. Each timeline-reference clock is maintained as a mapping from the core clock using the parameters tl_{drift} (drift correction), $core_{last}$ (the core-clock timestamp at the last synchronization event) and tl_{last} (timeline-reference timestamp at the last synchronization event). Using the current core timestamp, $core_{now}$, the timeline-reference time, tl_{now}, can be projected as follows:

$$tl_{now} = tl_{last} + tl_{drift} * (core_{now} - core_{last}) \quad (1)$$

A key proposition of Quartz is the ability to provide high-probability QoT bounds to applications. Therefore, every timestamp provided to applications has its QoT bounds appended to it. At any instant of time, the timing uncertainty ϵ is given by the following equation:

$$\epsilon = tl_{bound} + tl_{skew} * (core_{now} - core_{last}) \quad (2)$$

where, tl_{skew} is a high-probability upper bound on the drift of the timeline-specific clock, and tl_{bound} is a high-probability upper bound on the offset of the timeline-specific clock. Note that the probability of these bounds should be configurable by a system designer. Therefore, given a QoT accuracy requirement Q, the probability of the bounds being invalid can be given by $P(\epsilon > Q)$. Therefore, for each timeline clock, with high probability $1 - P(\epsilon > Q)$, we can say that a timestamp $tl_{now} \in [tl_{now} - \epsilon, tl_{now} + \epsilon]$.

Hardware Timestamping: Most modern network interfaces have their own clocks and also provide the ability to timestamp some or all network packets in hardware at the physical layer [16]. This enables both accurate packet timestamping and clock synchronization, and is referred to as hardware timestamping. Therefore, Quartz also supports network-interface clocks C_{net}, and maintains an accurate mapping between the core clock and network clock(s).

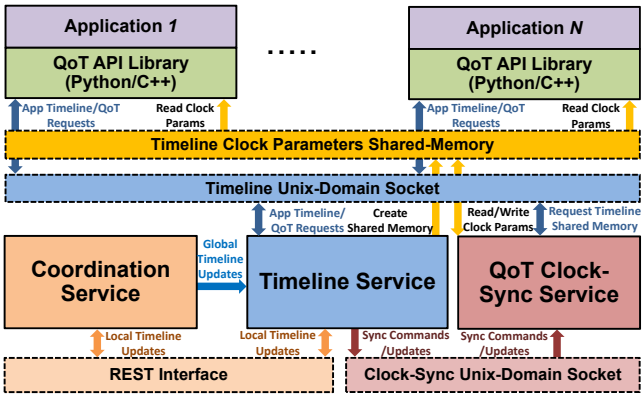


Figure 4: Quartz Time-as-a-Service. Solid boxes indicate components, dashed boxes indicate interfaces.

3.2 Quartz: Inner Workings

Figure 5 provides a global view of Quartz, which highlights its hierarchical architecture. We now describe the inner workings of the services and their interactions.

3.2.1 Facilitating Low-Latency Clock Reads From an application perspective, it is desirable that the timeline reference be read with low latency. To read a timestamp with its corresponding QoT, an application requires the current core-clock timestamp along with the timeline-projection and QoT parameters (Equations 1 & 2). Therefore, for each timeline, the timeline service creates a shared-memory region which holds the timeline projection and QoT calculation parameters. Applications can request to map this shared-memory region, with read-only privilege, into their own virtual-memory space. Thus, by reading the core clock and applying the timeline projection parameters from shared memory, an application can read the timeline reference with low latency. In Quartz, we choose the Linux real-time clock (`CLOCK_REALTIME`) as our core clock, as it is available on all Linux systems, and can be read with low latency from user space [35]. Applications obtain read-only access to the timeline-clock shared memory. This prevents malicious applications from modifying parameters held in shared-memory.

3.2.2 Handling Application Requests Quartz provides a library implementation of its API which helps applications make requests, and removes the complexity of directly interacting with the timeline service. The API calls made by the application are handled by the timeline service. However, only API calls related to (i) binding/unbinding from a timeline, (ii) updating timeline QoT requirements, and (iii) getting latency estimates between a pair of nodes, need to be handled by the timeline service. All other API calls related to

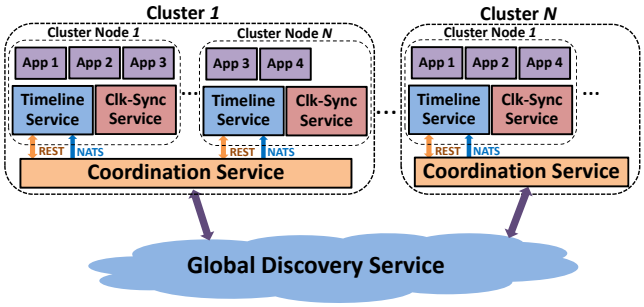


Figure 5: Quartz Time-as-a-Service at global scope

scheduling sensing/computation/actuation, and timestamping events are handled by the library in the context of the application. Quartz implements C++ and Python bindings. However, the API can be generalized to any programming language which supports socket programming and shared memory. We now describe how Quartz handles API requests.

Timeline Creation/Deletion: When an application binds to a timeline, the information is sent to the timeline service using the API. If the timeline does not exist on the node, the timeline service creates an instance of the timeline. This instance keeps track of all applications on the node bound to that timeline, and the instance is deleted when no active bindings exist. The timeline service also checks if the timeline exists at the coordination service (GET), and if not, it registers the timeline at the cluster scope (POST). If the timeline exists at cluster scope, then the timeline service updates (PUT) the QoT requirements, if they are more stringent than the timeline's most stringent existing QoT requirements. Similarly, the coordination service updates (creates) the timeline on the global discovery service at global scope if it exists (does not exist). A similar chain of events occurs for timeline deletion. The timeline service also creates a per-timeline shared-memory clock used to hold the timeline-projection and QoT-estimation parameters. This shared-memory region is passed to the clock-synchronization service, which updates the projection and QoT-estimation parameters to synchronize the local timeline clock to the timeline reference.

Event Timestamping: Since Quartz is designed for containerized applications, there are three types of possible events: (1) software events timestamped by the system clock, (2) network events timestamped by the system clock (software/kernel timestamping) or the network-interface clock (hardware timestamping), and (3) externally-timestamped events on a sensor. While events of type (1) and (2) are commonly observed in software systems, events of type (3) are most likely to be observed in embedded systems.

To support time-stamping software and network events, the clock-synchronization service maintains a mapping between the core and network clocks (if hardware timestamping is supported), along with the projection from the system (core) time to the timeline reference. Whenever these projection parameters are updated, the clock-synchronization service publishes them using NATS [36], which provides publish-subscribe-based communication. The API library subscribes to these projection parameters and maintains a ring buffer of the last n parameters. Based on the incoming event system/network timestamp, the Quartz API library chooses the appropriate parameters from the ring buffer to project the event timestamp to the timeline reference.

In some embedded systems with general-purpose I/O pins, some pins have the ability to detect a voltage-change event and record (or *capture*) a corresponding hardware-timer value. This voltage-change event can also be triggered by a sensor. Through appropriate transformations, this hardware-timer value can be mapped to a timestamp on a timeline. In Quartz, we expose all such timestamping hardware using the Linux **`ptp_clock`** [37] abstraction.

Event Scheduling: Scheduling an application on a *timeline* is important for executing distributed tasks/actuation synchronously. Therefore, the Quartz API library provides the ability to schedule events after a fixed time instant or duration on the timeline reference, in the form of *wait-until* calls, which suspend the application until a specified time instant. The library implements event scheduling internally, and schedules all events on the core clock. Therefore, the timeline-projection parameters are used to translate a scheduling request on the timeline reference to the core clock. Given that Quartz uses the Linux real-time clock (`CLOCK_REALTIME`), the Quartz API Library internally uses the existing `clock_nanosleep` POSIX API [24] to schedule computation/actuation on `CLOCK_REALTIME`.

3.2.3 Clock Synchronization and QoT Estimation Quartz features a flexible implementation allowing integration with multiple clock-synchronization protocols over IP-compliant networks. Our implementation utilizes NTP [15], PTP [16] and Huygens [26] clock-synchronization protocols, and avoids re-inventing the wheel. This is because existing protocols like NTP and PTP are well-tuned for modern hardware, and are based on standards and implementations that have evolved and been refined over time. Meanwhile, Huygens is a recently-proposed protocol for data centers [26].

However, unlike traditional clock-synchronization protocols which are best-effort, Quartz monitors the delivered QoT to check if it is within the application-specified limits, and orchestrates the synchronization-protocol parameters to meet them. We now describe how Quartz provides an *autonomous* clock-synchronization service, which dynamically responds to application QoT requirements as well as external changes (network disconnections, changes and load). Quartz is autonomous in the sense that, based on application requirements, it chooses an appropriate: (i) protocol, (ii) clock

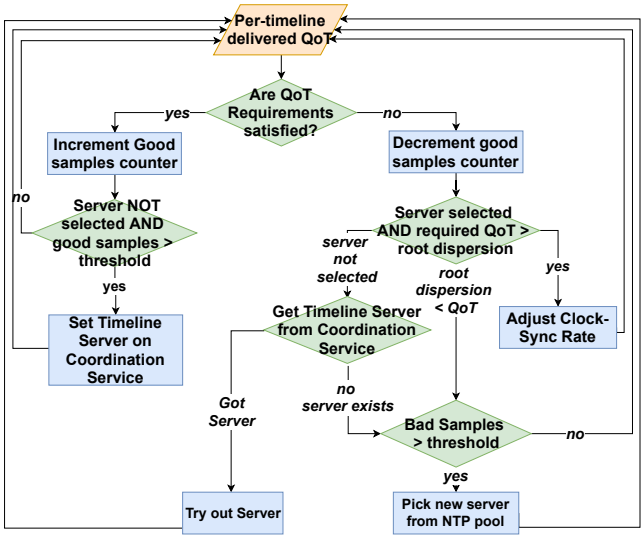


Figure 6: Adaptive NTP: Server Selection & Rate Adaptation

reference, and (iii) clock-synchronization tuning parameters. We first describe how Quartz synchronizes clocks for global timelines, and subsequently local timelines.

Global-Timeline Clock Synchronization: As global timelines can potentially span multiple clusters in different geo-distributed regions, the simplest way to maintain a shared notion of time is to synchronize all clocks to a common reference. To do so, Quartz uses the Network Time Protocol (NTP) [15] to synchronize the global timeline reference to Universal Coordinated Time (UTC). We use the *chrony* [40] implementation of NTP, which synchronizes the local clock by communicating with a set of NTP servers, and choosing the best source as the reference clock [40]. However, traditional NTP clients are often either configured using default or application/topology-specific tailor-made configurations. As Quartz is aware of application requirements, it responds to application demands by dynamically configuring NTP.

At startup, the Quartz timeline service automatically creates a default global timeline, and starts an NTP session on the synchronization service. Since, in our implementation, all global timelines follow a single clock reference, it suffices to maintain a single set of timeline-projection parameters for all global timelines. When an application binds to a global timeline specifying its timing requirements, Quartz checks if the application QoT requirement is being met. As all global timelines follow UTC, the QoT requirements of a global timeline are always defined with respect to UTC. Therefore, we use the *root dispersion* and *clock skew* values provided by NTP, which give a conservative estimate of how *far* or uncertain the clock is relative to UTC, to obtain a node's QoT. If the application QoT requirements are not being met, then Quartz tries to (i) either modify the synchronization rate, or (ii) if the root dispersion corresponding to the chosen reference indicates that the QoT requirements cannot be satisfied, it picks a new server from the pool of NTP servers. For a newly-created timeline, if the chosen server is able to deliver the desired QoT, Quartz registers this server with the cluster-specific coordination service, which in turn registers it with the discovery service. This allows other nodes on the same timeline, at both cluster and global scope, to select the same server as one of their reference sources. Thus, we ensure that nodes on the same timeline have a similar set of clock references. If available, Quartz also automatically chooses network hardware timestamping. Figure 6 presents a flow chart illustrating how we make NTP adaptive.

Local-Timeline Clock Synchronization: As local timelines are constrained to the cluster scope, we utilize PTP or Huygens (based on the system configuration) to synchronize the clocks in the cluster. When a local timeline is created on a node, the timeline service requests the synchronization service to start a timeline-specific synchronization session, and monitors the delivered QoT.

Precision Time Protocol (PTP) [16]: If PTP is the configured local-timeline protocol, then there is no need to choose a reference, as PTP automatically chooses a reference using the best-master clock-selection (BMC) protocol [16]. However, Quartz modulates the PTP synchronization rate (message-exchange frequency) in order to match the application QoT requirements to the delivered QoT. Quartz uses the *linuxptp* [41] implementation of the PTP standard.

Huygens [26]: is the state-of-the-art protocol well-suited to operate at cluster scale. Huygens uses a mesh of probes, which estimates the offsets between pairs of nodes using a Support Vector Machine (SVM). Based on the probe-mesh topology, the pair-wise offsets are sent to a centralized server, which uses the network effect to calculate the final node offsets with respect to a pre-defined in-cluster clock reference. As Huygens is not open-source, we have written our own implementation which consists of three major components: (1) per-node probe client-server pair which compute the pair-wise offsets and periodically publish the offsets, (2) per-cluster offset-calculator which calculates and publishes the final offsets by subscribing to the pair-wise offsets, and (3) per-node offset-receiver which subscribes to the final offsets.

If Huygens is the configured protocol, then the clock synchronization (1) topology, (2) rate and, (3) clock reference can be configured. Whenever a local timeline is created on a node, a *unique* offset-receiver is started per-timeline (allowing per-timeline clock-references), while the probe mesh and the offset-calculator are started at the first time a local timeline is created on the cluster. To meet the application-specified QoT, Quartz modulates the probe-mesh frequency, while monitoring the delivered QoT. In this initial version of Quartz, given that Huygens is designed to operate with a centralized server, we statically define the clock-synchronization

topology and the master reference. However, future extensions can make topology and master selection dynamic.

QoT Estimation: To estimate the QoT for local timelines using PTP or Huygens, the timing uncertainty relative to the local-timeline-reference needs to be computed by each node. Our implementation utilizes the methodology proposed in [42] to compute timing uncertainty. The proposed approach takes a sliding window of n samples of the clock frequency-drift and offset (computed by the clock-synchronization protocol). After estimating the distribution of their variances, it computes a high-probability upper-bound on the clock offset and the drift, which can be used to estimate the QoT (Equation 2). Both the number of samples n and the confidence probability of the bounds can be configured.

Adaptive Synchronization Rate: Unlike NTP, both PTP and Huygens are master-driven synchronization protocols. This means that the master node drives the synchronization rate. For example, in PTP, the master clock reference sends periodic multi-cast `SYNC` packets to all the slaves at a pre-determined rate. The slave nodes then respond with follow-up packets, and hence, the master controls the rate of clock synchronization. Note that, having a single rate for the entire network implies the node with the tightest QoT requirements holds significant influence on the synchronization rate. On the other hand, NTP is a client-driven protocol, and each client can independently decide and adapt its clock-synchronization rate by initiating a synchronization-request with an NTP server(s). Therefore, for both PTP and Huygens, we employ a similar clock-synchronization rate-adaptation strategy. Each node in a timeline periodically publishes its current delivered QoT on a particular timeline-specific topic using the NATS publish-subscribe mechanism. The master node listens to all the slave nodes, and tries to configure the synchronization rate to try to meet the QoT requirements of all the nodes on the timeline. Each master node has a protocol-specific lower and upper bounds on the rate of packets it can send. At the start, the master sends a burst of packets to quickly synchronize all the clocks. Subsequently, the master reduces its rate to the recommended protocol-specific rate. Based on whether the QoT requirements are being met or not, the master can gradually increase or decrease its synchronization rate.

Our implementation is open source and can be found at: *https://bitbucket.org/sandeepdsouza93/quartz/*. We now illustrate how *TimeCop* (Section 2) is deployed using Quartz.

3.3 Enabling TimeCop with Quartz

To demonstrate TimeCop, since we do not have ready access to real traffic controllers in a city, we simulate a city-scale traffic scenario with multiple intersections, using the open-source SUMO traffic simulator [28]. We use TraCI [28] to interface with the simulation, and ensure that each time-step in the simulation mirrors the flow of time in the real world. Using TraCI, we expose each intersection as MQTT [30] endpoints which (i) periodically publish intersection sensor state – the number of vehicles queued per-incoming lane in the last period, and (ii) listen for commands – the next phase of the traffic signals at the intersection. Note that using MQTT decouples the simulation logic from the controllers.

Each containerized intersection controller is deployed using the Nutanix Xi IoT [31] platform. Each controller gets the intersection state by subscribing to the MQTT endpoints corresponding to the intersection. The controller is based on deep reinforcement-learning [32], which uses the current intersection state to dynamically decide the next phase of the traffic signals at the intersection. The controller also periodically receives timestamped state from adjacent intersections, which it uses to improve traffic flow in coordination with other intersections. The chosen phase is published to the intersection MQTT endpoint listening for commands. Each intersection controller uses Quartz to bind to the *traffic-management* timeline with a QoT requirement of +/-1 ms, while Quartz ensures that all controllers bound to the timeline can meet their QoT specification.

The source code to build and deploy *TimeCop* can be found at: *https://bitbucket.org/sandeepdsouza93/traffic_app/*

4 Evaluation

We now evaluate the performance and scalability of Quartz. We first assess the accuracy of the clock-synchronization protocols that Quartz supports: NTP [15], PTP [16] and Huygens [26]. For these protocols, we consider different time-stamping options (hardware/software) and platforms. In particular, we consider two embedded/edge-form-factor platforms: Intel NUC [43] and Beaglebone Black (BBB) [44]. Secondly, we highlight the ability of Quartz to adapt to application-specific QoT requirements, and accurately estimate the delivered QoT. Lastly, we evaluate the scalability of Quartz by creating a geo-distributed-scale deployment.

To measure clock-synchronization accuracy we have setup a testbed consisting of Intel NUCs (dual-core Intel Core i3, 8 GB RAM) and Beaglebone Blacks (uni-core ARM Cortex-A8, 1 GB RAM) nodes. The two platforms are representative of both embedded (BBB) and edge-computing (NUC) devices. In terms of packet timestamping, the NUC supports hardware timestamping of all UDP packets and the BBB supports hardware timestamping of PTP-compliant packets.

The test-bed consists of two LANs: *greenwich* and *roseline*. Greenwich has two clusters (i) *NUC-Amethyst*: Kubernetes cluster with 4 NUCs, and (ii) *BBB-Citrine*: 4 BBBs with Docker; and an event generator *BBB-Onyx*. The event generator creates events (UDP packets, or voltage-change on a

hardware pin), which serve as opportunities for other nodes to timestamp. By comparing the timestamps of a common event, the offset between two clocks can be measured. Roseline has one cluster *BBB-Ametrine*: 4 BBBs with Docker.

We now describe each of the cluster types and their utility.

1) The **BBB Clusters** are used to benchmark the performance of (i) Huygens and PTP with hardware timestamping, and (ii) NTP with software timestamping. The BBB hardware strictly restricts hardware timestamping to PTP-compliant multi-cast packets sent/received on port 319 over 4 prescribed multi-cast IPs [47]. This constrains us to performing Huygens micro-benchmarks with not more than 4 nodes. However, the BBB have GPIO pins which allow 42ns-resolution timestamping on a rising or falling edge (generated by the event-generator BBB-Onyx). This allows us to externally measure the offset between two clocks and validate our implementation. In addition, having two BBB clusters on different LANs (citrine on greenwich, and ametrine on roseline) also enables clock-synchronization accuracy measurements between the two LANs.

2) The **NUC Cluster** is used to benchmark the performance of NTP, PTP and Huygens with hardware timestamping. The NUC features a desktop-class processor and a low-cost gigabit network interface [45] which supports hardware timestamping. As the NUC does not have external pins, the synchronization accuracy cannot be externally measured. Instead, we use the event generator (BBB-Onyx) to periodically generate multi-cast UDP packets, which the NUC timestamps in the network-interface hardware. We use these timestamps (after applying the timeline-projection parameters) to compute a safe upper bound on the clock offset.

4.1 Quartz: Clock-Synchronization Accuracy

We now evaluate the performance of NTP, PTP and Huygens in various scenarios based on (i) timestamping capability (hardware/software), (ii) platform (NUC/BBB), and (iii) server stratum (for NTP). Our micro-benchmarks are intended to provide a glimpse of the *best-effort* accuracy deliverable by a protocol on a given platform. This helps us to gain insights required to autonomously select an appropriate protocol and configuration within Quartz.

1) **NTP**: The NTP accuracy results are summarized in Table 2. In all the experiments, we utilize publicly-available NTP pool servers, and each node can pick its own server. This provides us with a good estimate of the accuracy achievable in real-world deployments without the need for custom NTP infrastructure. However, better accuracy can be achieved using custom NTP-server deployments. If we compare the measured accuracy based on platform type or timestamping capabilities, no significant differences are observed as (i) NTP requires few resources, and (ii) most NTP servers do not support hardware timestamping on their end. However, regardless of platform, the choice of server (stratum) plays an important in the accuracy obtained, as lower-stratum NTP servers track UTC with lower error. Thus, even choosing different stratum 1 servers can yield sub-millisecond accuracies across different LANs (Inter). Thus, NTP is well suited for global-scale applications which have QoT requirements in the order of 100s of μs to several ms.

Table 2: NTP [15] Accuracy (μs)

Platform	Timestamps	Cluster	Stratum	Max	Mean	Std. Dev
NUC	HW	Intra	1	4267	**380**	**633**
	HW	Intra	2	12607	2480	3351
BBB	SW	Intra	1	1638	**542**	**245**
	SW	Intra	2	5855	2380	717
	SW	Inter	1	2127	**929**	**553**
	SW	Inter	2	6033	3582	1032

Table 3: PTP [16] Accuracy (μs)

Platform	Timestamps	Rate (s)	Max	Mean	Std. Dev
NUC	HW	1	183	31	113
	HW	2	220	24	32
	HW	4	13	**9**	**2**
BBB	HW	1	14	**2**	**3**
	HW	2	39	8	7
	HW	4	39	5	7

2) **PTP**: Both platforms support hardware timestamping of IEEE 1588 PTP packets, and the accuracy results are summarized in Table 3. The network we utilize is not PTP-compliant and does not correct for queuing delays, which is mostly true for real-world networks. For both platforms, we observe that PTP at LAN-scale can yield accuracies in the order of 1-100 μs. This is primarily due to the use of hardware timestamping. On the NUC, decreasing the synchronization rate causes a slight increase in accuracy. In contrast, on the BBB, a lower synchronization rate yields marginally better accuracy. Thus, a faster rate does not always imply better accuracy. The Allan intercept of the clock [46], an indicator of clock stability, influences the optimal rate. Therefore, choosing the correct rate *autonomously* is useful in achieving application-specified levels of QoT.

3) **Huygens**: The accuracy micro-benchmarks for Huygens are summarized in Table 4. For both platforms, we consider a toy deployment of 4 nodes (in their clusters) with the probe-mesh pairs setup to form a 4-node loop. We observe that Huygens at LAN-scale can yield accuracies in the order of 100s of μs. The values in the table are for a pair of nodes separated by one hop in the probe mesh, while the values within parentheses are for a pair of nodes separated by two hops. Huygens relies on exchanging 10-100s of packets between nodes every second, and is designed for data-centers and not low-cost hardware. In both platforms, while using hardware timestamping, we observed significant

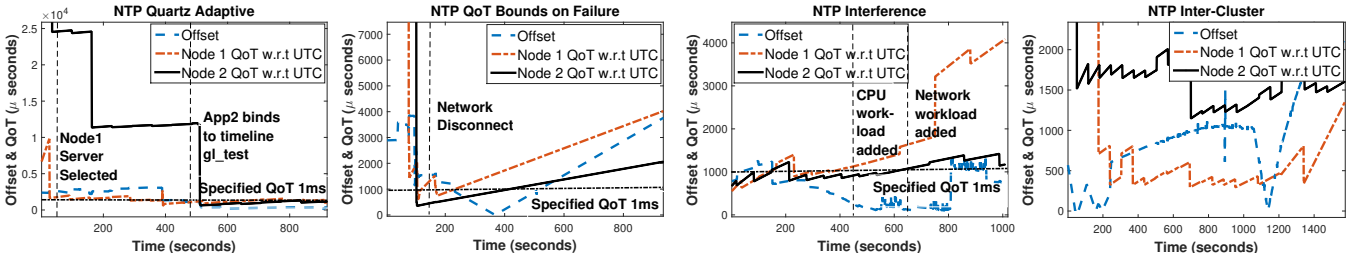


Figure 7: Quartz NTP: (a) Adaptive clock-synchronization, (b) QoT bounds on clock-synchronization failure, (c) Effect of CPU & network interference on QoT, and (d) Inter-cluster QoT estimation.

Table 4: Huygens [26] Accuracy (μs)

Platform	Timestamps	Rate (ms)	Max	Mean	Std. Dev
NUC	HW	10	401 (1596)	294 (1099)	21 (501)
	HW	100	405 (382)	**104 (105)**	**64 (75)**
	SW	10	1835 (1205)	294 (252)	242 (163)
	SW	100	1251 (965)	234 (328)	259 (243)
BBB	HW	100	13000000	2000000	3000000
	SW	10	782	**170**	**153**
	SW	100	4593	1091	340

timestamping errors at the network interface. This was especially severe for the BBB, which incorrectly orders/loses timestamps when packets arrive rapidly. Hence, we observe accuracies of the order of a few seconds, as the BBB NIC is only designed to timestamp PTP packets arriving at a rate of about 1-4 packets per second [47]. We also run Huygens with kernel timestamping, and observe that it yields an accuracy in the order of 100s of μs, and the synchronization is stable.

Therefore, while NTP and PTP are well-suited to run on low-cost platforms, Huygens needs more resources.

4.2 Quartz: Adaptiveness & QoT Estimates

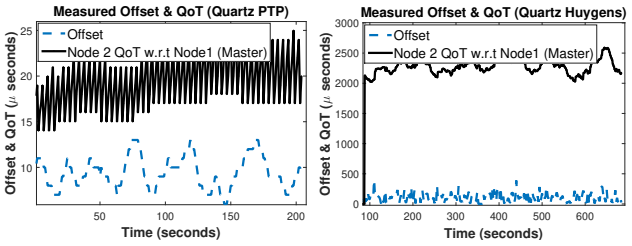
The key proposition of Quartz is to provide Time-as-a-Service, and adapt to application-specific QoT demands. We now evaluate Quartz's ability to: (i) orchestrate clock synchronization protocols to deliver application-specific QoT requirements, and (ii) report accurate QoT estimates to applications during transient (external disturbances) or permanent failure (network disconnect). We first focus on NTP, as it is our protocol of choice for providing TaaS at geo-distributed scale (global timelines). Subsequently, we benchmark PTP and Huygens for cluster-scale local timelines.

4.2.1 Global Timelines: Figures 7 (a)-(d) showcase four scenarios, where two application components α_1 and α_2, on two different nodes (Node1 and Node2), each bind to a global timeline *gl_test*, specifying their QoT requirements of +/-1 ms relative to UTC. Each figure plots the measured offset between the two nodes, as well as the QoT estimate that Quartz provides to each application. As the QoT bounds presented for global timelines are always relative to UTC, the bounds can be lesser than the measured offset between two nodes. As mentioned in Section 3.2.3, all global timelines are maintained relative to UTC, and hence, we use only one NTP instance to synchronize all the global timelines.

Adaptivity: Figure 7 (a) showcases Quartz's ability to adapt to application-specific QoT requirements. At time $t = 0$, α_1 on Node1 (NUC-Amethyst-1) binds to the timeline *gl_test*. As the existing clock reference cannot satisfy α_1's requirements, Node1's synchronization service tries new servers from the NTP pool, until the first dashed line, when it selects a suitable server which meets α_1's requirements. At time $t = 500$, α_2 on $Node_2$ (NUC-Amethyst-2) binds to *gl_test*. As Node2's current reference cannot satisfy α_2's requirements, Node2's timeline service queries the cluster-scope coordination service for any known NTP servers being used by other apps on *gl_test*. As a server exists (registered by α_1), Node2's synchronization service selects it, and is able to meet α_2's QoT requirements. Thus, the offset between the nodes reduces, and is reflected in the QoT bounds returned to α_2.

QoT-based Fault Detection: Figure 7 (b) plots a network disconnection scenario where clock synchronization is lost, as a node(s) is unable to communicate. At time $t = 180$, we simulate a network-disconnect/synchronization-service failure by killing the synchronization service on both Nodes1&2 (NUC-Amethyst-1&2). In this scenario, the API library (used by α_1 & α_2) uses the last-known QoT parameters to keep estimating the QoT (using Equation 2), until the clock is re-synchronized. As highlighted in Figure 7 (b), the bounds diverge linearly at the rate given by the upper bound of the clock drift (tl_{skew}). When the bounds exceed application-specified QoT requirements, the application is notified.

Resilience to CPU/Network Interference: Processing and networking resources are essential to clock synchronization. Figure 7 (c) illustrates the effect of adversarial CPU and network-intensive workloads on the QoT and offset between two nodes (NUC-Amethyst-1&2). Between time $t = 500$ and $t = 700$, we introduce a CPU-intensive workload on Node1 using the *stress* tool [48]. The *stress* tool creates 10 CPU-intensive threads which nearly saturate the CPU on node1. Observe that, as NTP is a lightweight protocol, there is no significant effect on the measured clock-synchronization



(a) (b)

Figure 8: Quartz QoT estimation: (a) PTP (b) Huygens

accuracy, and this is also reflected in the QoT bounds. At time $t = 700$, we introduce a network-intensive workload on Node1 using the *iperf* tool [52]. The *iperf* tool fully saturates the network interface on Node1 with TCP traffic. Shortly after the network load is introduced, there is a degradation in clock-synchronization accuracy, as reflected by the ~4x increase in the measured offset between Nodes1&2. Note that the QoT bounds delivered to the application on Node1 also suddenly increase to reflect this degradation in clock-synchronization accuracy. Hence, Quartz detects transient changes in QoT due to anomalies or interference.

Inter-Cluster QoT Estimation: Computing accurate QoT estimates across clusters in different LANs is key to providing TaaS at geo-scale. Figure 7 (d) plots the measured offset between two nodes (BBB-Citrine-1&BBB-Ametrine-1) in different LANs (*greenwich* and *roseline*), as well as the reported QoT. As the QoT is defined relative to UTC, for the bounds to be valid, the sum of the two QoT bounds should not be less than the measured offset between the two nodes.

4.2.2 Local Timelines : Figures 8 (a) & (b) plot the QoT estimates for local timelines, when using PTP and Huygens respectively. For both protocols, the QoT and offset of Node2 are defined relative to the timeline reference (Node1). Both sets of measurements were obtained using a pair of NUCs (NUC-Amethyst-1&2). For Huygens, we observed significantly higher QoT bounds than the measured offset. This is due to the high variance in the clock offset and drift measurements caused by hardware timestamping instabilities/errors in the network interface. Therefore, for local timelines, we focus on the PTP protocol. We utilize a pair of nodes in the *BBB-Citrine* cluster to perform experiments. As stated before, the BBB have GPIO pins which allow 42ns-resolution timestamping of a voltage-change event (generated by the event-generator BBB-Onyx). We use this to externally measure the offset between two clocks.

Adaptivity: Figures 9 (a) & (b) showcase Quartz's ability to orchestrate PTP to adapt to application QoT requirements. The left y-axis shows the measured offset and the estimated QoT, and the right y-axis shows the binary logarithm ($\log_2$) of the period of the PTP SYNC messages [16]. As described in section 3.2.3, Quartz modulates PTP's clock-synchronization rate to meet application QoT requirements. We consider α_1 on Node1 (BBB-Citrine-1) and α_2 on Node2 (BBB-Citrine-2) bound to the local timeline *test*. In both Figures 9 (a) & (b), α_1 on Node1 is elected as the timeline master-clock reference, and the application QoT requirements are set to (a) 10μs and (b) 5μs respectively. In Figure 9 (a), Quartz initially increases the rate to quickly meet the QoT requirements, and then slows down once the QoT requirements are met. Similar observations can be made for the case illustrated in Figure 9 (b). Note that for a multi-cast protocol like PTP, decreasing the synchronization rate can lead to significant reduction in network bandwidth consumed. Additionally, in case (b), sometimes during durations of high-synchronization rates, there can be timestamping instabilities, which cause the offset, and the delivered QoT to spike. We believe that this is due to an issue in the BBB hardware timestamping.

Table 5: Continental-scale Scalability Results

Specified QoT (Accuracy)	**Worst Delivered QoT**	**Best Delivered QoT**
500μs	442μs	284μs
1ms	994μs	233μs

QoT-based Fault Detection: Figure 9 (c) plots a network disconnection scenario where clock synchronization is lost, as a node(s) is unable to communicate. At time $t = 280$, we simulate a network-disconnect failure by killing the synchronization service on Node2 (BBB-Citrine-2). Similar to the NTP case, the API library (used by α_2) uses the last-known QoT parameters to keep estimating the QoT (using Equation 2), until the clock is re-synchronized. As highlighted in Figure 9 (c), the bounds are diverged linearly at the rate given by the upper bound of the clock drift (tl_{skew}).

Resilience to CPU/Network Interference: Figure 9 (d) illustrates the effect of CPU and network-intensive workloads on Quartz PTP. At time $t = 200$, we introduce a CPU-intensive workload on Node2 using the *stress* tool [48] for 100 seconds, which fully saturates the CPU on the BBB. Since PTP is lightweight, there is no significant effect on the clock-synchronization accuracy, and the observed QoT bounds. At time $t = 400$, we introduce a network-intensive workload on Node2 using *iperf* [52] for 100 seconds. This saturates all the network bandwidth, and PTP packets cannot get through. Thus, the clock offset and the observed QoT diverge. As soon as the network interference goes away, Quartz increases the PTP clock-synchronization rate to ensure that the delivered QoT quickly returns to the desired level (10μs).

Therefore, Quartz adapts to application demands and external interference at both cluster and global scales.

4.3 Scalability

We now demonstrate Quartz's ability to provide Time-as-a-Service at geo-distributed scale, by utilizing clusters

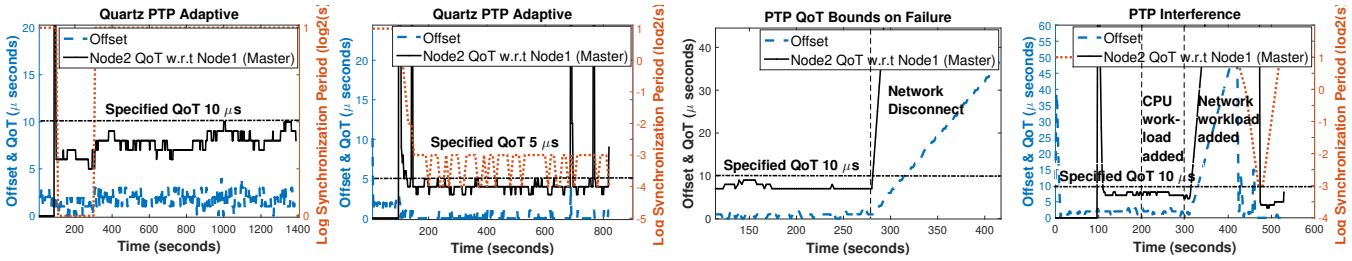


Figure 9: Quartz PTP: Adaptive clock-synchronization with QoT requirement (a) 10μs, (b) 5μs, (c) QoT bounds on clock-synchronization failure, and (d) Effect of CPU & network interference on QoT.

Table 6: Geo-distributed Scalability: Microsoft Azure

QoT Spec.	Region	Worst QoT	Best QoT	Average QoT	Fraction
500μs	east-us	506μs	200μs	327μs	0.98916
	central-us	504μs	216μs	354μs	0.98844
	west-europe	508μs	249μs	415μs	0.97398
	east-australia	NA	NA	NA	NA
	east-asia	NA	NA	NA	NA
1 ms	east-us	635μs	199μs	365μs	1
	central-us	568μs	140μs	293μs	1
	west-europe	640μs	307μs	476μs	1
	east-australia	1003μs	490μs	758μs	0.99076
	east-asia	1006μs	459μs	645μs	0.97398

Table 7: Geo-distributed Scalability: Google Cloud

QoT Spec.	Region	Worst QoT	Best QoT	Average QoT	Fraction
500μs	asia-east	716μs	230μs	376μs	0.96025
	asia-south	886μs	214μs	390μs	0.94606
	us-west	501μs	184μs	289μs	0.99850
	europe-north	389μs	186μs	291μs	1
	south-america	1100μs	276μs	473μs	0.87861
1 ms	asia-east	648μs	292μs	426μs	1
	asia-south	813μs	237μs	484μs	1
	us-west	1009μs	224μs	542μs	0.99566
	europe-north	509μs	204μs	309μs	1
	south-america	746μs	277μs	458μs	1

deployed using Virtual Machines (VMs) hosted in the public cloud. Our experiments are meant to demonstrate scale, and hence we consider global timelines maintained using Quartz's Adaptive NTP clock-synchronization protocol. As we cannot externally measure synchronization accuracy inside the VMs, we rely on the ability of our system to accurately provide QoT estimates, to check if different application-specified QoT levels can be achieved across all the geo-distributed clusters. We conduct two sets of experiments:

(1) **Continental Scale:** We deploy Quartz across 15 VMs running across three Amazon Web Services (AWS) [49] regions spanning the continental United States (5 VMs each in us-east-1 Virginia, us-east-2 Ohio and us-west-2 Oregon). Each VM is configured as a standalone Kubernetes cluster using the Nutanix Xi IoT [31] platform, which also helps deploy the Quartz micro-services as Kubernetes pods. In this experiment, we deploy an application with 15 coordinating components, each running in one of the 15 VMs. Each application component binds to a common global timeline `gl_test`, and specifies its QoT requirement. This experiment gives us an idea of the accuracy that Quartz can achieve at continental scale, for a geo-distributed deployment on a single network backbone. We conducted this experiment over a period of 5 hours and considered two QoT-specification levels (required clock-synchronization accuracy): 500 μs and 1 ms. For a given specified QoT level, Table 5 summarizes the best and worst QoT level delivered by Quartz across the 15 geo-distributed clusters. As seen in Table 5, the best QoT represents the tightest accuracy bounds observed, and the worst QoT represents the loosest bounds observed.

(2) **Global Scale:** We deploy Quartz across 20 VMs spanning five continents and two public cloud providers. Our deployment consists of 10 VMs running in five Microsoft Azure (Azure) [50] regions (2 VMs each in east-us, central-us, europe-west, australia-east and asia-east), and 10 VMs running in five Google Cloud (GCP) [51] regions (2 VMs each in asia-east, asia-south, us-west, europe-north and south-america-east). Each VM is configured as a standalone Kubernetes cluster. In this experiment, we deployed an application with 20 coordinating components, each running in one of the 20 VMs. Each application component binds to a common global timeline `gl_test`, and specifies its QoT requirement. This experiment was used to assess the accuracy that Quartz can achieve for a geo-distributed deployment. We conducted this experiment over a period of 5 hours and considered two QoT-specification levels (required clock-synchronization accuracy): 500 μs and 1 ms. Tables 6 and 7 summarize the best, worst and average observed QoT, along with the fraction of time the specified QoT requirements were satisfied, for the VMs deployed in Azure and GCP respectively.

For our continental-scale experiments on AWS (Table 5), we observe that Quartz can reliably deliver an accuracy level of 500μs. On the other hand, for our global-scale deployment across Azure and GCP (Tables 6 and 7), we observe that some nodes cannot achieve a QoT level of 500μs. This is especially true for the Azure nodes deployed in the east-australia and east-asia regions (values indicated by *NA* in Table 6). This is because Quartz is unable to choose an appropriate NTP server to satisfy the QoT specification of 500μs.

The lowest-possible uncertainty with respect to UTC, achievable by a client using a specific NTP server, depends on the

server's: (i) stratum [15], i.e., how *closely* it tracks UTC, and (ii) the round-trip network latency between the server and the client. For example, if Quartz chooses a low-stratum server located in the United States (US), which tracks UTC accurately, then the high round-trip latency between the server in the US and a client in Australia will constitute the dominant factor in the clock-synchronization uncertainty or QoT reported by Quartz. This may prevent the QoT requirements of 500 μs from being met.

Thus, Quartz maintains global timelines with sub millisecond accuracy, while estimating QoT at geo-distributed scale.

5 Related Work

We now present a survey of relevant prior work.

Clock Synchronization: The utility of a shared notion of time in distributed systems has been well-studied in prior work. In [13], the benefits of using synchronized clocks in distributed systems was analyzed. The author concluded that synchronized clocks can improve performance by replacing communication with local computation [13]. In the context of model-based design, PTIDES [22] is a hardware-software co-design framework to model, design and deploy time-critical embedded applications, using a shared notion of time. For safety-critical systems in the automotive and aerospace domains, the Time-Triggered Architecture (TTA) [23] provides a deterministic way to deploy systems using a shared clock. However, both PTIDES and TTA are designed for the embedded domain, and cannot scale to geo-distributed cyber-physical applications which run in distributed heterogeneous environments including the cloud and the edge.

There has also been some work on distributed-programming idioms that support time as a first-class citizen. Examples include Stampede [53] which uses application-specified "virtual time" as the basis for enabling temporal causality in distributed applications, Stampede-RT [54] which allows distributed applications to timestamp events with real-time tags, and Persistent Temporal Streams [55] which unifies in-memory and stable storage temporal events of a given activity. However, these systems do not expose timing uncertainty to applications, which is key to providing time-as-a-service.

Google's geo-distributed Spanner database utilizes synchronized clocks with the uncertainty information to achieve global-scale consistency [3]. However, Spanner is a closed system, is not adaptive and relies on dedicated infrastructure. Additionally, the TrueTime API [3] is tailored only to database transactions and does not treat the notion of QoT as an application-specified requirement.

Clock-synchronization technologies such as GPS, Network Time Protocol (NTP) [15], and Precision Time Protocol (PTP) [16], can achieve a reliable and accurate shared notion of time. Most recently, a number of protocols [25][26] have been proposed to achieve nanosecond-accuracy clock synchronization in data-centers. Notable among these is Huygens [26] which uses a peer-to-peer probing mesh along with Support Vector Machines (SVM) to compute clock offsets between nodes. However, all the above mentioned protocols are best-effort and do not consider application-specific QoT requirements. Therefore, most systems using these protocols end up being over-engineered to meet the needs of pre-determined applications. Hence, there is a need for an application-level framework which can respond to application timing demands, while making it easy to develop time-based distributed applications.

Timelines and the QoT Stack: In [1], the authors introduced the *timeline* abstraction, which abstracts away clock-synchronization from applications. The authors also introduced the QoT Architecture [1] along with its kernel-space realization for Linux, called the QoT Stack. Subsequently, the work in [18] proposed QuartzV, which adds para-virtual extensions to the QoT Stack, to provide near-native timing performance for applications running in virtual machines. However, the QoT Stack utilizes kernel-space components, which significantly limit both its portability and scalability.

6 Conclusion

Time is a key primitive for enabling coordination in distributed systems. In this work, we introduced Quartz which provides *Time-as-a-Service* to geo-distributed containerized applications, which coordinate using a shared notion of time. Based on the notion of Quality of Time (QoT) in conjunction with the timeline abstraction, Quartz exposes an API which simplifies the development of geo-distributed coordinated applications, and allowing applications to specify their QoT requirements. Quartz orchestrates the underlying infrastructure to meet these application-specific requirements, and exposes the delivered QoT back to the application. Thus, time-based distributed-coordination applications can be fault-tolerant in the face of clock-synchronization failure.

Quartz features a micro-service architecture. This makes it scalable and easy to deploy on platforms ranging from embedded devices to the edge, and the cloud. Our evaluation indicates that Quartz adapts to application demands, and maintains a *timeline* across multiple geo-distributed nodes. Future work will look at improving Quartz's security properties, as well as supporting admission control based on application requirements and system capabilities.

While Quartz is most relevant for CPS, the core concept of Time-as-a-Service is also useful for distributed software applications, such as databases and logging systems. We strongly believe that the ability to request and observe QoT can be used to relax many of the stringent asynchronous assumptions associated with distributed systems.

Acknowledgements

The authors would like to thank the anonymous reviewers for their constructive feedback, and our shepherd Umakishore Ramachandran for his help and guidance. The authors would also like to thank our collaborators at Nutanix Inc. for supporting this work, and particularly Govardhan Reddy Jalla for setting up some of the experiments. This research is funded in part by the National Science Foundation under award CNS-1329644. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, or the U.S. Government.

References

[1] F. Anwar, S. D'souza, A. Symington, A. Dongare, R. Rajkumar, A. Rowe and M. Srivastava, "Timeline: An Operating System Abstraction for Time-Aware Applications", in Proc. of *IEEE Real-Time Systems Symposium*, 2016

[2] B. Regula, "Formation control of a large group of UAVs with safe path planning", in Proc. of *IEEE Mediterranean Conference on Control and Automation*, 2013.

[3] J. C. Corbett et al., "Spanner: Google's globally distributed database", in *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, 2013

[4] S. Natarajan and A. Ganz, "SURGNET: An Integrated Surgical Data Transmission System for Telesurgery", in *International Journal of Telemedicine and Applications*, Article ID 435849, 2009

[5] J. Enright and P. Wurman, "Optimization and Coordinated Autonomy in Mobile Fulfillment Systems", In Proc. of *AAAI Workshop*, 2011

[6] M. Buevich, X. Zhang, D. Schnitzer, T. Escalada, A. Jacquiau-Chamski, J. Thacker and A. Rowe, "Microgrid Losses: When the Whole is Greater Than the Sum of Its Parts", In Proc. of *7th IEEE/ACM International Conference on on Cyber-Physical Systems*, 2016

[7] SAE J2735 Standard, *https://ntl.bts.gov/lib/51000/51100 /51167/DE156E CC.pdf*

[8] R. Rajkumar, I. Lee, L. Sha and J. Stankovic, "Cyber-Physical Systems: The Next Computing Revolution", in Proc. of *Design Automation Conference*, 2010

[9] F. Bonomi, R. Milito, J. Zhu and S. Addepalli, "Fog computing and its role in the Internet of Things", in Proc. of the *MCC workshop on Mobile cloud computing*, 2012

[10] P. Simoens et. al, "Scalable Crowd-Sourcing of Video from Mobile Devices", CMU-CS-12-147 Tech Report, 2012

[11] M. Satyanarayanan et al., "The Case for VM-Based Cloudlets in Mobile Computing", in *IEEE Pervasive Computing*, Vol. 8, Issue: 4, Oct.-Dec. 2009

[12] C. Hung, G. Ananthanarayanan, P. Bodík, L. Golubchik, M. Yu, V. Bahl and M. Philipose, "VideoEdge: Processing Camera Streams using Hierarchical Clusters", in *ACM/IEEE Symposium on Edge Computing*, October 2018

[13] B. Liskov, "Practical Uses of Synchronized Clocks in Distributed Systems", in Proc. of *ACM symposium on Principles of distributed computing*, 1991

[14] T. Mizrahi and Y. Moses, "Serving time in the cloud: Why time-as-a-service?", in Proc. of *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2016

[15] D. L. Mills, "Internet time synchronization: the network time protocol," in *IEEE Transactions on Communication*, vol. 39, no. 10, 1991.

[16] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "IEEE 1588-standard for a precision clock synchronization protocol for networked measurement and control systems", in *IEEE Instrumentation and Measurement Society Standard*, 2005

[17] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize everything but time", in Proc. of *OSDI*, 2010

[18] S. D'souza and R. Rajkumar, "QuartzV: Bringing Quality of Time to Virtual Machines", in Proc. of *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2018

[19] S. D'souza and R. Rajkumar, "Time-based Coordination in Geo-Distributed Cyber-Physical Systems", in Proc. of *USENIX Workshop on Hot Topics of Cloud Computing*, 2017

[20] Docker Containerization Platform, *https://www.docker.com/*

[21] S. D'souza and R. Rajkumar, "A Cyber-Physical OS for Enabling Spatio-Temporal Coordination at Geo-distributed Scale", in Proc. of *Workshop on Next Generation OS for Cyber-Physical Systems (NGOSCPS)*, 2019

[22] J. Zou, S. Matic, E. Lee, T. Feng and P. Derler, "Execution Strategies for PTIDES, a Programming Model for Distributed Embedded Systems", in Proc. of the *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009

[23] H. Kopetz, "The time-triggered architecture", in *Proc. of the IEEE*, Vol. 91, No. 1, January 2003

[24] M. Kuhn, "Implementing POSIX clocks under Linux", *https://www.cl.cam.ac.uk/ mgk25/posix-clocks.html*

[25] K. Lee, H. Wang, V. Shrivastav, and H. Weatherspoon, "Globally synchronized time via datacenter networks", in *Proc. of the 2016 conference on ACM SIGCOMM*, 2016

[26] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization", in *Proc. of NSDI*, 2018

[27] J. Gertler, "Analytical Redundancy Methods in Fault Detection and Isolation", in *Preprints of IFAC/IMACS Symposium on Fault Detection, Supervision and Safety for Technical Processes*, 1991

[28] Simulation of Urban Mobility (SUMO), *http://sumo.dlr.de/index.html*

[29] SUMO Traffic Control Interface, *http://sumo.dlr.de /wiki/TraCI*

[30] MQTT connectivity protocol, *http://mqtt.org/*

[31] Nutanix Xi IoT Platform, *https://www.nutanix.com/products/iot*

[32] V. Mnih et al., "Human-level control through deep reinforcement learning", in *Nature*, 518.7540 (2015): 529

[33] Kubernetes, *https://kubernetes.io/*

[34] Apache Zookeeper, *https://zookeeper.apache.org/*

[35] VDSO - overview of the ELF shared object, *http://man7.org/linux/man-pages/man7/vdso.7.html*

[36] NATS Pub-Sub, *https://nats.io/*

[37] PTP hardware clock infrastructure for Linux, *https://www.kernel.org/doc/Documentation/ptp/ptp.txt*

[38] Docker: Add host device to container, *https://docs.docker.com/engine/reference /commandline/run/add-host-device-to-container—device*

[39] Ping, *https://linux.die.net/man/8/ping*

[40] Chrony NTP, *https://chrony.tuxfamily.org/*

[41] The Linux PTP Project, *linuxptp.sourceforge.net/*

[42] A. Bondavalli, F. Brancati, and A. Ceccarelli, "Safe estimation of time uncertainty of local clocks", in *Proc. of International Symposium on Precision Clock Synchronization for Measurement, Control and Communication*, 2009

[43] Intel NUC Kit NUC7i3BNK, *https://www.intel.in/content /www/in/en/products/boards-kits/nuc/kits/nuc7i3bnk.html*

[44] Beaglebone Black, *https://beagleboard.org/black*

[45] Intel Ethernet Connection I219-V, *https://downloadcenter.intel.com/product/82186/Intel-Ethernet-Connection-I219-V*
[46] D.W. Allan, "Clock Characterization Tutorial", *https://tf.nist.gov/general/pdf/2082.pdf*
[47] AM335X CPSW Ethernet Driver Guide, *processors.wiki.ti.com/index.php/AM335x_CPSW_(Ethernet)_Driver27s_Guide*
[48] Stress, *https://linux.die.net/man/1/stress*
[49] Amazon Web Services, *https://aws.amazon.com/*
[50] Microsoft Azure, *https://azure.microsoft.com/*
[51] Google Cloud Platform, *https://cloud.google.com/*
[52] Iperf Network Measurement Tool, *https://iperf.fr/*
[53] U. Ramachandran, R. S. Nikhil, J. M. Rehg, Y. Angelov, A. Paul, S. Adhikari, K. M. Mackenzie, N. Harel, and K. Knobe, "Stampede: A Cluster Programming Middleware for Interactive Stream-Oriented Applications", in *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, 2003
[54] D. Hilley and U. Ramachandran, "Stampede RT: Programming Abstractions for Live Streaming Applications", in *Proc. of International Conference on Distributed Computing Systems (ICDCS)*, 2007
[55] D. Hilley and U. Ramachandran, "Persistent Temporal Streams", in *Proc. of ACM Middleware*, 2009