CROSS City Cloud: Location-Certification-as-a-Service

Lucas de Haan Vicente
Instituto Superior Técnico, Universidade de Lisboa, Portugal
lucasvicente@tecnico.ulisboa.pt

***Abstract*—Lisbon is one of the world's most visited cities, attracting millions of tourists each year and many of them use smartphone applications to discover points of interest. Although these applications heavily rely on location information, most of them are susceptible to location spoofing. Location certificates can be used to thwart these attacks.**

***CROSS City* is a smart tourism application that rewards users for completing tourist itineraries and one of its strengths is the use of location certificates. The location verification relies on the periodic collection of wireless network observations by multiple users to make sure the travelers went to the tourist attractions.**

In this work, we introduce *CROSS City Cloud*, a cloud-native and improved location certification system, capable of producing and validating time-bound location proofs using data collected from publicly available Wi-Fi network infrastructure. The architecture was extended to efficiently compute the stable and transient networks of a given location required to determine location and time-of-visit. We deployed to the Google Cloud Platform, including an additional control plane to ease service operation. The smart tourism application was utilized to demonstrate the feasibility of our Location-Certification-as-a-Service. The merits of the solution were validated with performance evaluations and real-world scenario assessments stressing each component of the system in various aspects, such as its scalability and feasibility.

***Index Terms*—Location Spoofing Prevention, Location Proof, Context-Awareness, Security, Internet of Things, Cloud Deployment.**

I. Introduction

Modern mobile applications and services rely heavily on location to provide users with context-aware information. Practical use cases of these services include: *map navigation*, *smart tourism*, *weather services* and *location-based games*. Several techniques can be used to provide location context to applications. However, many of these services do not verify the location information they consume, making them vulnerable to various location spoofing attacks [1]. To combat and provide protection against these attacks, location proof systems [2]–[4] provide a means for producing reliable digital certificates attesting an individual's presence at a geographical location and specific time. The generated certificates can subsequently be utilized to validate location claims.

An initial version (v1) of CROSS[1] City [5] was developed for a smart tourism use case; tourists use their smartphones to interact with existing infrastructure at points of interest in the city. They periodically collect data. And in the end, rewards are awarded for completing tours, which in turn motivates them to continue using the application. However, rewards also entice bad actors to illegitimately obtain them. To combat this, CROSS offers multiple strategies for producing location proofs, namely, scavenging of Wi-Fi identifiers, one-time codes broadcast by Wi-Fi beacons, and user interaction with kiosks. Robust attestation for the time of visit requires the use of the mentioned Wi-Fi beacons. To reduce the need for this additional infrastructure, further improvements can be performed on the scavenging strategy [6]. The improved location proof generation process relies on data collected from smartphone sensors and its analysis against previously obtained Wi-Fi data from a given location, to proof visits with temporal granularity. CROSS City v1 lacks the necessary components and protocols to properly support these enhancements. Moreover, the initial version of CROSS was a prototype not designed with production properties in mind, such as reliability, availability, scalability, and performance. It also lacks an automatic approach for deploying its various components. CROSS v1 does not offer health status of the back-end or a way to determine the cause of failures. If all the aforementioned issues were solved, we would be close to providing a Location-Certification-as-a-Service platform.

In this paper we propose *CROSS City Cloud*, a cloud-native location certification system with support for time-bound location proofs, serving as a testbed to demonstrate the feasibility of embedding location certification into public cloud computing technology to provide novel Location-Certification-as-a-Service capabilities.

We will provide background and details on the implementation, as well as the results of the experiments.

II. Background

Maia et al. [5] proposed CROSS (v1) that implements a set of location proof techniques for consumer mobile applications. While doing itineraries through points of interest in the city, tourists interact with Wi-Fi and other existent infrastructure using their mobile devices which records traces of information (*trip logs*). Three entities are defined, prover (makes a claim with location evidence), witness (endorses claims with their collected evidence), verifier (analyzes evidence and makes the decision to issue - or not - a location certificate).

CROSS v1 uses a client-server model consisting of a mobile application and a centralized server with a database

[1]The *CROSS* designation refers to loCation pROof techniqueS for consumer mobile applicationS

component. The server is responsible for handling the validation of location evidence submitted by the tourists. The server contains a persistent module with domain data such as user information, points of interest, tourism routes, estimated rewards and the set of Wi-Fi Access Points (identified by their SSIDs) expected to be present at each location.

CROSS v1 is able to employ three distinct strategies for location verification. Each strategy provides different levels of security by trading off infrastructure needs and operational costs:

- *Scavenging*: User collected Wi-Fi traces compared against the list of known networks at that location. Has a reduced setup cost, however it provides a weaker level of confidence, since an attacker could forge a trip log after knowing the list of networks;
- *TOTP*: Leverages a Time-based One-time Password similar to the proposed in RFC 6238 [7], in the broadcast SSID. Requires the deployment of a customized Wi-Fi AP dynamically changes its broadcast SSID in a set period. Only the Wi-Fi AP and the CROSS server share a secret which is used to produce and validate the codes. Attests for both a user's location and visiting period, at the expense of setup cost;
- *Kiosk*: Clients produce location proofs by interacting with a trusted kiosk device, which signs the information later verified by the server. This strategy is more inconvenient for the clients and requires an extra infrastructural component.

A. Time-Bound Location Proofs Based On Scavenging

Aligned with the CROSS application, Claro et al. [6] collected a dataset of Lisbon *hotspots*, as well as developed a *data model* and algorithms to determine the location and time interval of a tourist visit. More specifically, their approach leverages diverse ad-hoc witnesses to observe *long-lived hotspots* and *short-lived hotspots* to detect the location and prove the time of visit, respectively, of other tourists. A prover's location claim uses as evidence a collected set of Wi-Fi Access Point SSIDs, referred to in the model as *observations*. Three time windows are defined in the model to bound the observations for verification:

- *Epoch*: The most encompassing time frame. Only observations collected within this time window will be used to compute the stability of the Wi-Fi networks at each location for the following *epoch*;
- *Period*: A subdivision of an *epoch*. Only observations collected within this time window will be used to compute the volatility of the Wi-Fi networks at a given location;
- *Span*: A subdivision of a *period*, a *span* is the interval formed by the time of visit in the location claim (t_p) and an additional parameter *delta* (δ) between $t_p - \delta$ and $t_p + \delta$. Prover and witness must share observations in the same span.

B. Data Processing Architectures

Now that we have a model to implement, we studied possible architectures. Most data-sensitive systems require real-time data analytics. These systems require asynchronous data transformations with minimal delay without sacrificing processing of historical data, meaning to reprocess past input data, to cope with change. Lambda and Kappa are two widely utilized architectures to address the challenges outlined above.

1) Lambda Architecture: Proposed in 2011 [8] with the goal of achieving both real-time and historical data processing capabilities by combining both batch and stream methods. The Lambda architecture, as illustrated in Figure 1, is composed of three distinct layers: a *batch* layer, a *speed* layer and a *serving* layer. Data is fed to both the batch and speed layers. The batch layer stores the immutable master dataset and recomputes a series of *batch views* that facilitate the computation of arbitrary queries over the dataset. Running batch jobs to maintain these precomputed views of the dataset takes a significant amount of time. Furthermore, any queries made during this process lack access to the data captured during this period. Therefore, to compensate for high latency updates the speed layer is responsible for incrementally computing a series of *realtime views* of recent data. Queries are handled by the serving layer against the merged results of both the batch and realtime views [9]. These capabilities increase the complexity of code maintainability, since two separate processing systems need to be indefinitely maintained. Both are also required to be synchronized to correctly integrate the missing updates that occur during a batch job [10].

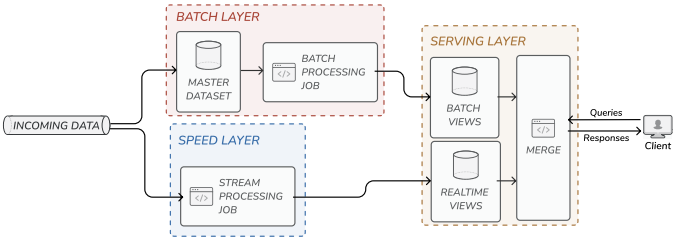

Fig. 1: Overview of the Lambda data processing architecture.

2) Kappa Architecture: Proposed in 2014 [11] to overcome the limitations of the Lambda Architecture. In the Kappa architecture there is no notion of batch, every data is treated as a stream and therefore only a stream processing engine is required. As illustrated in Figure 2, it consists of two distinct layers: a *stream* processing layer and a *serving* layer. Data is fed as streams to the stream processing layer which is responsible for running the real-time data processing jobs and then queries are handled by the serving layer against these results. It is important to note, that data may still be reprocessed by simply *streaming* through historical data. This architecture achieves a "*general-purpose*" solution with both real-time and reprocessing capabilities without the added complexity of maintaining two separate systems by trading-off latency/throughput and efficiency when reprocessing historical data [10].

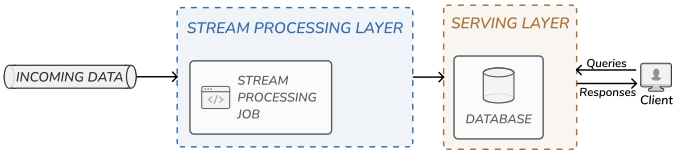

Fig. 2: Overview of the Kappa data processing architecture.

III. Architecture

We now discuss the extension made to the CROSS data management layer to ingest, aggregate and integrate scavenged network observations for time-bound location proof validation.

A. Data Management Layer Architecture

Before starting a specific trip, the client application fetches the catalog of possible itineraries. During the trip, the client application logs the visits to each point of interest while sensing Wi-Fi signals, and either stores these locally (offline connection) or immediately publishes them to the API server (online connection). At the end of the trip, the application submits all of the collected information, to the back-end and claims each point of interest visited. Based on this flow, we could either opt for a minimal Lambda architecture with solely batch processing, sufficient to provide us offline data processing capabilities or a Kappa architecture ensuring both offline and real-time data analytics. Real-time capabilities allow us to relax the assumptions that the start and end of a trip coincide with the beginning and end of a *period*, and that during the *period* no location claim request is made for that particular *period*. Solely with offline or batch processing, any location proof requests made during a *period* would necessarily have to be purposely delayed until a *period* reached its completion or multiple high latency batch jobs would have to be triggered during the *period*.

To avoid this, we extend the CROSS architecture data management layer based on the *Kappa Architecture*, with three distinct layers: a *domain* layer, a *stream* layer and a *serving* layer. The domain layer is responsible for storing and handling queries related to entity relation data, such as the user information, points of interest and tourism routes. The stream layer stores the raw streams of Wi-Fi signal observation data as atomic and immutable facts, kept as the truth within a given *epoch* time window, through the use of publish time timestamps, which is useful for recomputing views historically. Additionally, the stream layer is responsible for executing stream processing jobs that produce stream views containing precomputed aggregated results to assist stable or volatile set queries. The serving layer indexes and uses the precomputed results, received from the stream layer, to serve the stable or volatile set query requests. Figure 3 illustrates the resulting CROSS City server-side architecture.

B. Collection of Network Observation Data

Networks collected by tourists must be continuously integrated onto the operational dataset, nonetheless to ensure that only valid network observations are integrated, the CROSS

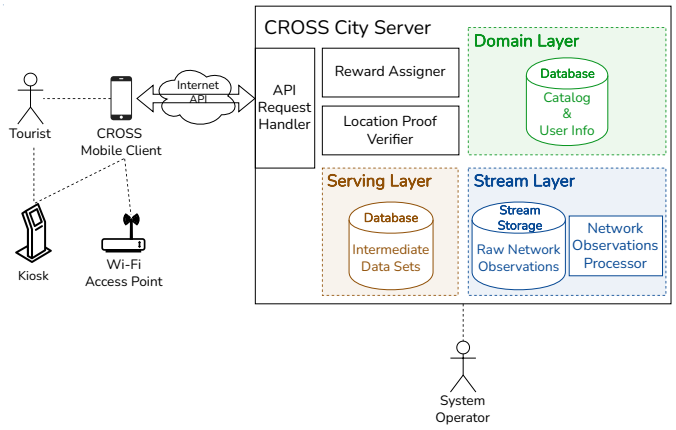

Fig. 3: Overview of the extension to the CROSS server-side architecture.

City Cloud lifecycle, illustrated in Figure 4, is split into two stages: *Pre-Live* and *Live*. The Pre-Live stage is a finite time interval with a total duration equal to the system *epoch*, referred to as $epoch_0$. Throughout the Pre-Live stage, only trustworthy entities such as the system operators submit network observations of each existing point of interest, with the goal of deriving the initial/genesis stable network sets. Since the system trusts system operators, verification of their submissions is therefore not required. The Live stage is a sequence of *epoch* intervals, with the initial one named $epoch_1$. Throughout the Live stage, untrusted entities interact through trip submissions. Tourists/Provers are meant to complete trips across multiple point of interest visits and collect the networks observed. Each visit (location claim) contains a set of evidences (network observations) which are validated against the claimed point of interest stable network set of the former epoch. For $epoch_n$, the stable network set of $epoch_{n-1}$ is used for validation. Only if the claimed location confidence threshold is fulfilled does the visit get accepted and its network observations submitted to be integrated in subsequent stable and volatile network sets.

C. Computation of Intermediate Observation Set

The stream layer's sole goal is to produce network observation views to be queried efficiently, in a low-latency manner. Hence, we are adopting an *incremental computation* approach over *recomputation*, avoiding the execution of our function logic over the entire set of observations. To be efficient, the views should contain intermediate results of the expected queries: *Most observed networks, over an epoch, for a given point of interest* (stable network set) and *Least observed networks in a span interval, over a period, for a given point of interest* (volatile network set).

The key idea of the intermediate views is to maintain a count of the number of observations per network at each point of interest, for the most encompassing range of time. Note that the usage of higher time intervals increases query performance by trading-off proof validation accuracy. We now reason about

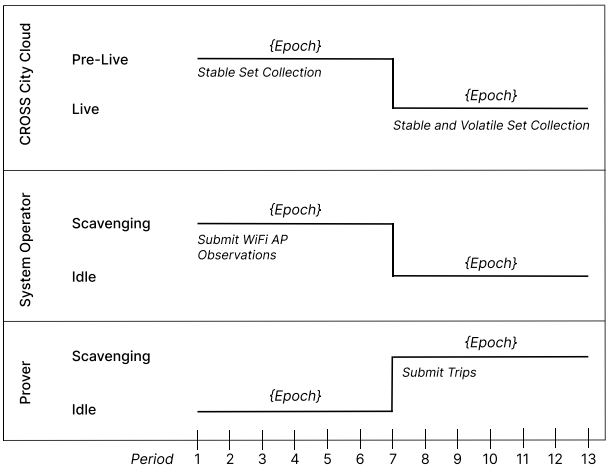

Fig. 4: UML timing diagram of the CROSS City Cloud lifecycle across $epoch_0$ and $epoch_1$ with the system operator and prover entity lifelines.

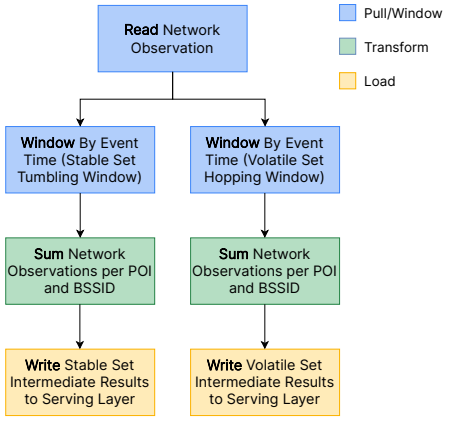

Fig. 5: CROSS City Cloud pipeline for producing intermediate stable and volatile network sets.

the ideal time interval for each set and the kind of window used for grouping (tumbling for fixed size non-overlapping time intervals and hopping for fixed size scheduled overlapping intervals [12]). Stable network sets are queried within an *epoch*; since an *epoch* must always encompass a *period*, the minimum time window granularity is a *period*. Volatile network sets are queried within a *span*, and each of the possible span time windows encompasses smaller intervals of span's greatest common divisor size. For example let the $spans = \{15\,min, 10\,min, 5\,min\}$, given any *span* interval with size equal to one of the *spans*, it can be represented as a union of 5 min (the greatest common divisor) intervals. We leverage the fact that the set of spans are known before going live to compute every possible time interval of span's greatest common divisor size with minute granularity, during a *period*, by aggregating network observations into one minute periodic hopping time windows. This computation is feasible as it results in, at most, 1440 windows per span interval (1440 minutes in a day) during computation time. Figure 5 represents the pipeline's DAG[2], each network observation is first pulled from the stream layer storage component, then aggregated in two separate tumbling and hopping windows, based on its publish time (event time), with size equal to the *period* and the greatest common divisor of the *spans*, then keyed and summed per point of interest and BSSID, and finally written to the serving layer.

D. Computation of Stable and Volatile Observation Set

Both stable and volatile network set views are persisted in the serving layer and partitioned by point of interest and *period*, since the queries are expected to be tied to a particular point of interest and require, at most, a *period* worth of data. Hence, the usage of the horizontal partitioning method is an efficient way to store these specific views. Each record in the view maintains the number of observations for a particular network within a time interval.

As soon as an *epoch* is completed the *period* intermediate stable set views, that comprise the *epoch*, are used to produce a materialized view containing the top 10% observed networks over that *epoch*, off the critical path. Network observations from past *epochs* are expected to remain unchanged, thus the creation of an additional materialized view significantly improves the efficiency when accessing a stable set. Furthermore, volatile set queries utilize the intermediate volatile set views and the stable set materialized view to filter the top 10% observed networks of the previous *epoch* and retrieve the bottom 10% observed networks within the claimed time interval. We are considering 10% as the threshold value.

IV. Cloud Deployment

We now discuss the deployment selection for each component of the three layers, based on the set of requirements it must fulfill. Figure 6 details the deployed cloud architecture on the Google Cloud Platform (GCP).

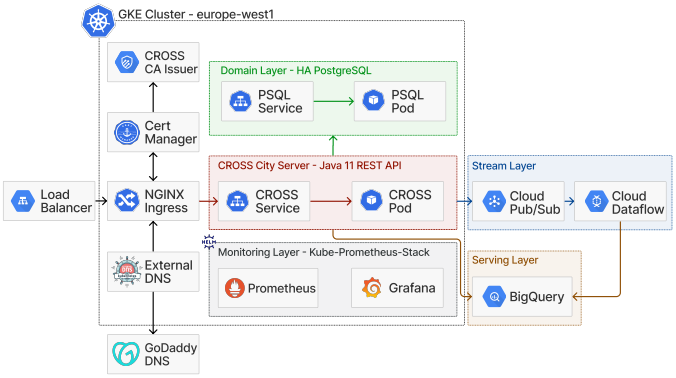

Fig. 6: Overview of the CROSS City Cloud Google Cloud Platform Architecture.

[2]Directed Acyclic Graph

A. Domain Layer

The domain layer is comprised of two primitive components: compute (API Server) and database (Domain Database).

The API Server is the client entrypoint for the CROSS City services, and is mainly responsible for managing user sessions, handle trip submissions and the respective location claims for reward attribution. The interface is implemented in Java with Maven[3] as the build tool, and follows the REST (REpresentational State Transfer) software architectural style for web services. The client communicates with the API server over HTTPS exchanging payloads defined and encoded with protocol buffer[4]. Any existent FaaS or PaaS cloud compute offering, such as GCP Cloud Functions or App Engine, AWS Lambda or Elastic Beanstalk and Azure Functions or App Service, is able to serve this kind of RESTful service. Nonetheless, to mitigate cloud provider lock-in, Kubernetes-based cloud services, such as Google Kubernetes Engine (GKE), offer managed orchestration tools as a service providing a complete control over every aspect of container orchestration, from networking, to storage, and observability over each component.

The database component of the domain layer is responsible for storing both user and tourism related data. User information is used to serve the authentication service, and provide specific information of each user account, regarding their trip history and rewards received. Tourism information is comprised of the available tourism routes, points of interest and possible rewards, referred to as the *catalog*. We maintained the use of a relational data model and PostgreSQL. The deployment of the database could be attained through fully managed services such as Google Cloud SQL, Amazon RDS and Azure Database for PostgreSQL. However, considering the use of GKE for the deployment of the CROSS API server, we can leverage cluster multi-tenancy as a means to be more cost effective, while maintaining a similar level of management.

B. Stream Layer

The stream layer is comprised of two primitive components: data ingestion and data processing.

The data ingestion component has the purpose of ingesting observation events, published by clients through the API, for streaming into the data processing component. To increase the level of tolerable faults and persist the streams, this component is a message broker avoiding direct communication between producers and consumers. Several options are available such as RabbitMQ[5], ActiveMQ[6], Apache Kafka[7], and Google Cloud Pub/Sub[8]. Nonetheless, the message broker must support multiple producers and consumers on the same topic. Message ordering is not necessary, since our intermediate results are aggregated on event-time not dependent on their delivery order. Replaying previous events is required, thus message retention with period of an *epoch* is necessary. To avoid the loss of any network observations and duplicates from retries, *at least once* delivery paired with *exactly once* processing semantics must be guaranteed. To handle a diversity of consumers, a Pull-based model where the consumer pulls messages from the broker is preferable over a Push-based model. Google Cloud Pub/Sub fulfills these requirements.

The data processing component, responsible for processing the network observations to produce the intermediate results for the stable and volatile set queries. The pipeline is meant to aggregate network observations on event-time in two separate tumbling and hopping windows - stable and volatile set window. Late and duplicate network observations should be expected. Aggregations should be performed on event-time, not processing-time. State and resource management should be automatic ensuring fault-tolerance and elastic scalability. Additionally, since the data ingestion component solely assures *at least once* delivery, we had to develop a way to guarantee *exactly once* processing semantics. Processing engines that satisfy the described needs include Apache Spark[9] and Apache Flink[10]. Nonetheless, to be engine agnostic the Apache Beam[11] open-source framework for parallel, distributed data processing at scale can be used. This allows us to plug into other execution engines, such as Apache Flink, Apache Spark or Google Cloud Dataflow. Google Cloud Dataflow is a fully managed service for executing Apache Beam pipelines, in Google Cloud, that fulfills our needs. Dataflow assures *at least once* semantics, by default, however in order to guarantee *exactly once* semantics sources and sinks must produce deterministic results. Deterministic outcomes allows the engine to deduplicate unacknowledged transformations that are retried. We ensure determinism through the use of deterministic unique IDs and specific I/O[12] APIs provided by the services used.

C. Serving Layer

The serving layer is composed of a single primitive component: database. It is accountable for persisting the aggregate intermediate results computed by the stream layer and serve query requests related to the stable and volatile signal sets. Both queries make use of a SUM aggregate function over the intermediate results network observations count, and either filter the resultant top 10% or bottom 10% observed networks, for the stable and volatile set queries, respectively. The database engine query language should allow us to express all of this query logic directly through it. To guarantee proper inter-layer operability and connectivity the database should be easily integrated with both the data processing stream layer component (Google Cloud Dataflow, detailed in Section IV-B) and the API domain layer component (Kubernetes pod with the Java REST server, described in Section IV-A). Writes are

[3]https://maven.apache.org/
[4]https://developers.google.com/protocol-buffers
[5]https://www.rabbitmq.com/
[6]https://activemq.apache.org/
[7]https://kafka.apache.org/
[8]https://cloud.google.com/pubsub
[9]https://spark.apache.org/
[10]https://flink.apache.org/
[11]https://beam.apache.org/
[12]Input/Output

expected to be made in real-time, so the database component must support streaming records to it, and reads may be performed randomly. Additionally, the database must scale as the size of the intermediate results increases and be fault-tolerant. Based on these requirements, the most suitable cloud service candidates are Google Bigtable (Key-Value - NoSQL) and Google BigQuery (Relational - SQL). Both services are fully managed with scalability, high availability and fault-tolerance ensured by either service. We decided to utilize Google BigQuery mainly due to its support of ANSI-standard SQL, granting us a higher level of expressiveness, and the seamless Google Dataflow integration for streaming records through the *Storage Write API*[13] with *exactly once* semantics.

D. Monitoring Layer

The monitoring layer collects, aggregates, and analyzes metrics to aid the system operator understanding the behavior of the system. Metrics are a measurement of a system at a given point in time that are intended to provide a picture of the system's health. This layer increases the level of observability, and ensures analysis capabilities over each aspect of CROSS City Cloud. Observability refers to the degree to which a system can be understood from its external outputs, such as CPU and memory utilization, disk space, latency, etc. Analysis refers to the activity of inspecting each observable data and retrieving useful information from it. To achieve this set goal, the monitoring stack used is comprised of Prometheus[14] and Grafana[15].

E. CROSS City Certificate Authority

Due to the fact that CROSS City Cloud is meant to be deployed to a public cloud provider, it is crucial to secure network communications between clients and the server, and provide a method for server authentication. The usage of HTTPS[16] protects the privacy and integrity of data transmitted while it is in transit. HTTPS authentication necessitates the use of a trustworthy third party to sign server-side digital certificates, hence we deployed to the Kubernetes cluster a CROSS City project private CA[17]. The deployment is achieved through *cert-manager*[18] a X.509 certificate controller for Kubernetes workloads with the purpose of handling the certificate management. The private CA is represented as an Issuer Kubernetes resource[19], able to issue signed certificates by fulfilling CSRs[20]. The cluster's Kubernetes Ingress resource is secured through the request of TLS signed certificates with additionally configured manifest annotations.

[13]https://cloud.google.com/bigquery/docs/write-api

[14]Prometheus is an open source, metrics-based monitoring system employed to collect and store real-time metrics as time series data - https://prometheus.io/

[15]Grafana is an open source analytics and interactive visualization web application, which allows us to query data stored in Prometheus - https://grafana.com/

[16]HTTP over TLS

[17]Certificate Authority

[18]https://cert-manager.io/

[19]https://cert-manager.io/docs/concepts/issuer/

[20]Certificate Signing Requests

V. Evaluation

After deploying our solution, we needed to validate it and assess its performance with demanding request workloads. We split the evaluation in two parts: the feasibility of providing location and time-bound proofs, and the performance assessments.

A. Experimental Setup

All of the performed evaluations, detailed in Section V-B, Section V-C and Section V-D, follow the same setup procedure. The selected benchmark tool alongside the test scripts are first wrapped up into a Docker image, tagged, and pushed to our google cloud project's container private registry. Then, the selected benchmark tool and CROSS City Cloud are deployed to distinct Google Kubernetes Engine (GKE) clusters, comprised of one and two nodes, respectively, physically isolated, in the europe-west1 region. A comprehensive specification of the GKE clusters used is detailed in Table I. It is worth noting that no resource limit was configured at any point to avoid resource contention when performing the tests.

B. Stable and Volatile Set Match as Location and Time Proof

To the feasibility of our solution in providing location and time-bound proofs, we used the real-world collected network observation data of the *LXspots* dataset [6]. Each point-of-interest has touristic relevance and different characteristics, such as being outdoors or indoors, sparse or central, and central or remote - Alvalade, Comércio, Gulbenkian, Jerónimos, Oceanário and Sé. Each smartphone - one Samsung Galaxy S9, one Huawei Mate 10 and one LG V10 thinq - represents a distinct prover - Alice, Bob and Charlie. Each prover stays at each point-of-interest for 15 minutes. We will consider the seven consecutive day *epoch* from 2019-07-29 to 2019-08-04 and the one day *period* of 2019-08-19.

1) Stable Set Match as Location Proof: To proof presence at the location, the prover's collected network observation set is compared against the set of stable networks, throughout a previous *epoch*, at the claimed point-of-interest. Table II presents the percentage of match between these two sets. Considering a *50%* match threshold to determine successful proof, all provers visits are attested at four out of the six locations (except for Alice in Sé). Given the total 18 visits, this equates to a stable set success rate of 61.11%. Stable sets produced through our solution seem viable to attest presence at a location. Locations lacking stable networks such as Jerónimos and Comércio, due to their characteristics, would favour from either a lower match threshold or the deployment of known Access Points (TOTP).

2) Volatile Set Match as Time Proof: To proof the visiting period, the prover's scavenged network observation set is compared against the set of volatile networks, throughout the *span*, at the claimed point-of-interest. The match percentage between these two sets is shown in Table II. The 15 minute visit is split into four span intervals - 15, 5, 3 and 1 min. Considering a *50%* match threshold to determine successful proof, visiting period attestation was achieved for all locations

TABLE I: Evaluation Google Kubernetes Engine clusters specification.

Stack	Machine Family	Machine Type	OS	Kernel Version	CPU	RAM Size	Disk Size	Disk Type	Additional Disk Size	Additional Disk Type	Network	Docker Ver	Kubernetes Ver	Region	Locations
CROSS City Cloud	General--Purpose	e2--highcpu--4	Container--Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (4)	4 GB	20 GB	Standard persistent disk	-	-	Default	20.10.12	1.21.11--gke	europe--west1	europe--west1-b, europe--west1-c
k6 Benchmark Tool	General--Purpose	e2--highcpu--8	Container--Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (8)	8 GB	20 GB	Standard persistent disk	200 GB	Regional Standard Persistent Disk	Default	20.10.12	1.21.11--gke	europe--west1	europe--west1-b
Real-Time Client	General--Purpose	e2--highcpu--8	Container--Optimized OS	5.10.107	Intel Xeon or AMD EPYC Rome @ 2.00+ GHz (8)	8 GB	20 GB	Standard persistent disk	200 GB	Regional Standard Persistent Disk	Default	20.10.12	1.21.11--gke	europe--west1	europe--west1-b

TABLE II: Prover's Stable and Volatile Set Match Percentage for each Point-of-Interest. (percentage $\geq 50\%$ in green and $< 50\%$ in red)

Point--of--Interest	Prover	Stable Set Match	Stable Set Success Rate ($\geq$ 50.00%)	Volatile Set Match for Claimed Span Interval				Volatile Set Success Rate ($\geq$ 50.00%)
				15 min	5 min	3 min	1 min	
Alvalade	Alice	100.00%	61.11%	100.00%	87.50%	100.00%	90.00%	63.89%
	Bob	100.00%		0.00%	61.53%	50.00%	58.33%	
	Charlie	92.85%		0.00%	30.76%	62.50%	46.15%	
Comercio	Alice	27.77%		20.00%	50.00%	0.00%	100.00%	
	Bob	30.55%		57.14%	100.00%	100.00%	100.00%	
	Charlie	27.77%		80.00%	0.00%	100.00%	100.00%	
Gulbenkian	Alice	100.00%		0.00%	12.50%	50.00%	91.66%	
	Bob	60.70%		54.54%	41.66%	33.33%	46.15%	
	Charlie	100.00%		44.44%	50.00%	78.57%	83.33%	
Jeronimos	Alice	9.30%		30.00%	50.00%	60.00%	75.00%	
	Bob	27.90%		9.09%	30.00%	25.00%	40.00%	
	Charlie	20.93%		54.54%	50.00%	33.33%	100.00%	
Oceanario	Alice	85.00%		83.33%	100.00%	100.00%	100.00%	
	Bob	65.00%		0.00%	50.00%	50.00%	60.00%	
	Charlie	75.00%		16.66%	14.28%	14.28%	50.00%	
Se	Alice	43.00%		60.00%	86.00%	33.33%	100.00%	
	Bob	50.00%		62.50%	66.60%	50.00%	75.00%	
	Charlie	54.00%		0.00%	33.33%	50.00%	75.00%	

in at least 50.00% of the provers' claimed span intervals. Most notably, 75.00% of the claimed intervals in Comércio and Sé were successfully attested. Given the total 72 claimed intervals, this equates to a volatile set success rate of 63.89%. Our solution's volatile sets seem to be effective in attesting for the visiting period. It is also important to note, that longer intervals have a lower success rate than shorter intervals - 15 min (44.44%) and 1 min (83.33%). Due to the user's network scan collection period of 30 seconds, we can view a longer interval's volatile set as a union of several shorter intervals' volatile sets. This demonstrate a higher dependence on consistent witness coexistence for long visiting period proof, since ideally network observations are scavenged equally, meaning by the same number of witnesses, throughout the full duration of the interval. Nonetheless, touristic visits are typically performed in specific groups and scheduled intervals, thus the feasibility of the solution remains plausible.

C. Domain Layer Scalability and Performance

To assess the performance and scalability of the domain layer, comprised of the API and database deployed to a cloud environment, we synthesized a test workload based on expected user access patterns. We will focus on the submission of trip visits through the domain layer as this is the typical path of a user's interaction with CROSS. The integration of scavenged network observations through the stream and serving layer is done asynchronously and will be evaluated separately. The workload is comprised of two stages with distinct load duration and ramping user concurrency. Each stage executes an identical user flow: the users sign in, retrieve existing routes, fetch a specific route, and submit a visit to one of the route's point of interest with a sufficient amount of Wi-Fi AP evidences to achieve the route's waypoint set confidence threshold (75%), as to claim that location.

Three separate configurations - *A*, *B* and *C* - of the domain layer were evaluated. The baseline configuration is comprised of a single replica (*A*), while the test configurations vary from one to two replicas (*B*) and from one to four replicas (*C*). Since preliminary tests demonstrate that the workload is expected to be CPU bound, each configuration horizontally auto-scales based on a pre-set average CPU utilization threshold (*40%*), using the Kubernetes Horizontal Pod Autoscaler resource, which provisions identical API server pods (*replicas*) to accommodate a growing work demand.

During the execution of the workload utilizing the k6[21] benchmark tool, the request response time, rate of requests and both the CPU and memory usage metrics were collected.

1) Scalability Model: Before delving into the experimental results and discussion, we briefly present the scalability model used in the experiment. The *Universal Scalability Law* (USL) [13] (Equation 1) extends *Amdahl's law* [14] (Equation 2) with an additional parameter (κ), allowing us to model capacity degradation related to coherency losses. Other scalability models exist such as the Exponential, Geometric and Quadratic. The USL, on the other hand, differs from the other models in that it is defined in terms of two parameters rather than a single one, accounting separately for both contention (serial work) and coherence (crosstalk among workers in the system such as nodes, CPUs, threads, etc). The contention component (σ) of the system ends up limiting asymptotically its speedup, while the coherence portion (κ) limits the maximum system achievable size.

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1) + \kappa N(N-1)} \quad (1)$$

$$X(N) = \frac{\lambda N}{1 + \sigma(N-1)} \quad (2)$$

[21] k6 is an open source load testing tool - https://k6.io/

TABLE III: *Universal Scalability Law* (USL) parameters for each configuration.

Configuration	λ	σ	κ
A	23.24	0.0409	0.0007583
B	23.57	0.0000	0.0006959
C	26.87	0.0000	0.0003925

X = throughput
N = concurrent users
λ = performance coefficient
σ = serial portion
κ = crosstalk factor

2) System Performance and Scalability: Using the throughput measurements collected per level of user concurrency, we are able to model the system's scalability with the *Universal Scalability Law*. Table III summarizes the resultant performance coefficient and scalability parameters estimations for each configuration, with their respective model plots in Figure 7. By comparing the estimated performance coefficients (λ) at the unitary load, we can quantify the efficiency of the system across sizes. Doubling the size of the system from both configuration *A* to *B* and *B* to *C* we maintain approximately 51% and 57% efficiency, respectively. Despite the efficiency values being lower than expected, these can be explained by the fact that all configurations start the test workload execution with the same amount of replicas (1), and only when the scaling policy is met do configurations *B* and *C* provision further resources. Regarding the scalability of the system, the maximum useful user concurrency of each configuration, calculated through the Equation 3 in relation to both scalability parameters, is 35, 37 and 50 users for *A*, *B* and *C*, respectively. Based on the maximum useful user concurrency, the maximum speedup achieved between configuration *A* (247 req/sec) and *C* (684 req/sec) is of approximately $2.77x$.

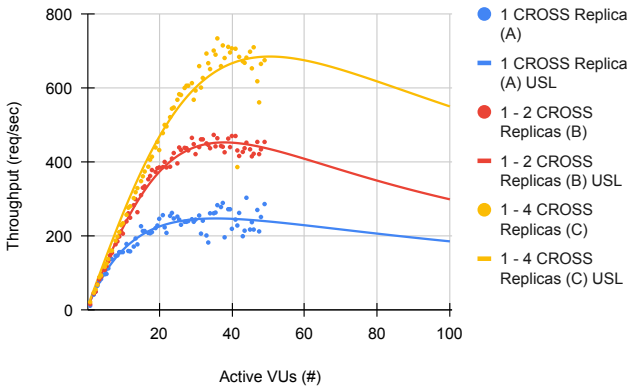

Fig. 7: Throughput over Active Concurrent Virtual Users for each system size configuration and respective USL model.

$$N_{\max} = \frac{\sqrt{1-\sigma}}{k} \tag{3}$$

3) Request Performance: From a request performance practical standpoint, derived from the collected mean latency measurements plotted in Figure 8, we note that for a latency threshold of *100 ms*, both configurations *A* and *B* are only able to perform below set threshold solely until 17 and 37 concurrent virtual users, respectively, after which a high level of degradation is noticeable. Despite an outlier peak at 41 concurrent virtual users, configuration *C* is capable of performing below set threshold for the full duration of the test workload. Furthermore, in an effort to further quantify the overall user experience and perception of the system with each configuration, we have plotted the percentile 90 latency (filters top 10% worse latencies) in Figure 9. Percentiles are useful for us to determine the expected maximum response time for a percentage of requests/users. In this particular case, only configuration *C* is able to able to withstand a set latency threshold of 200 ms (double the set mean latency threshold), meaning we are able to conclude that for 90% of users, within the tested range of user concurrency, will experience a response time either as fast or faster than 200ms.

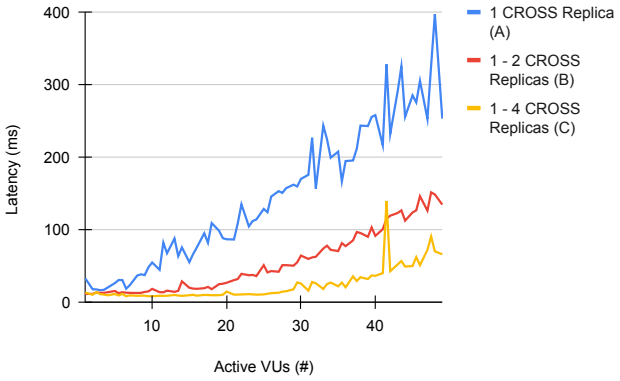

Fig. 8: Mean Latency over Active Concurrent Virtual Users.

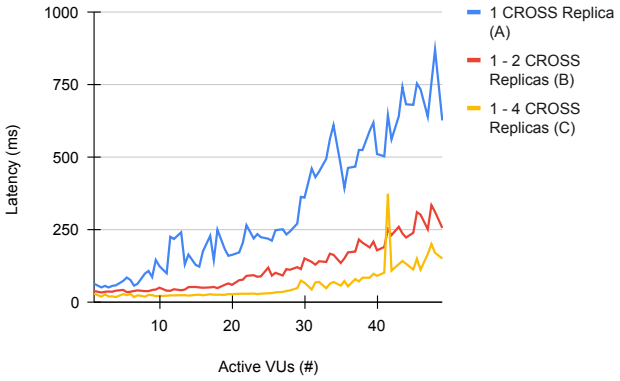

Fig. 9: Percentile 90 Latency over Active Concurrent Virtual Users.

As expected configuration *C* reaches a higher level of resource utilization on both CPU (A - 2.38 cpus, B - 3.90

cpus and C - 4.79 cpus) and memory (A - 2.06 GiB, B - 2.37 GiB and C - 3 GiB), albeit significantly within the cluster limits of 8 cpus and 8 GiB.

4) Summary: For this specific synthetic workload, configuration *C* is capable of outperforming the other two configurations with regards to both system and request performance metrics, as observed and predicted. Moreover, we infer that the system is able to scale horizontally, while maintaining an acceptable level of performance and resource utilization.

D. Stream Layer Performance and Completeness

To estimate the trade-offs made in performance, completeness and cost of the unbounded Apache Beam pipeline solution, a scenario was setup consisting of real-time submissions of a dynamic set of network observations of a specific point of interest as Wi-Fi AP evidences, simulating both user collection and submission. During the execution of the test workloads a set of metrics were collected including the rate of observations processed, and the data watermark lag which refers to the amount of time since the most recent output watermark - network observation publish time. CPU and memory usage were also monitored, but will not be further detailed.

Note that, as detailed in Section IV-B, Dataflow assures *at least once* semantics by default, however in order to guarantee *exactly once* semantics modifications had to be performed to ensure that both the Pub/Sub source and the BigQuery sinks were deterministic. As this factor (the level of correctness) is expected to have the most impact on the performance of this layer, we quantified its impact by comparing both the pipeline's processing semantics.

1) Impact on Throughput: In Figure 10, the plot shows a speedup, of approximately $3.5x$, with *at least once* semantics all throughout the pull/window phase in which each stage performs consistently at the same rate. Nonetheless, as noted in the transform phase plot in Figure 11 the performance gains obtained in the pull/window phase with *at least once* semantics are lost, and both semantics end up performing at identical levels (approximately $1.0x$ speedup), indicating a potential bottleneck at these stages. In the load phase (Figure 12), with *at least once* semantics the pipeline is able to sustain the rate of observations from the previous phase, confirming the bottleneck suspicion with these semantics, as opposed to the pipeline with *exactly once* semantics which is not able to maintain the rate at this phase, more specifically at the *Write to Big Query* stages (the bottleneck stages with these semantics), where we deduct a $165x$ speedup.

2) Impact on Output Data Watermark: The data watermark lag, as previously explained, is the age up to which all data has been processed by the pipeline, which allows us to quantify the processing time in relation to the publish time of an observation (the client collection time). In this specific test workload, observations can be delayed at most 1 minute, thus the expected data watermark lag is at least 1 minute plus the cumulative processing time in the API server, Pub/Sub and the pipeline. We observe that as expected both pipelines have a data watermark lag of at least 1 minute, and in spite of

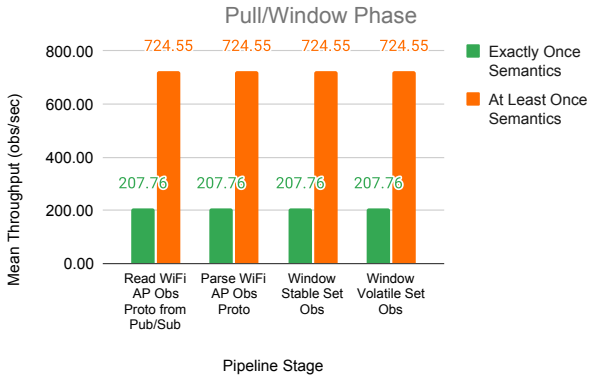

Fig. 10: Mean throughput per pipeline stage in the pull/window phase for each semantics.

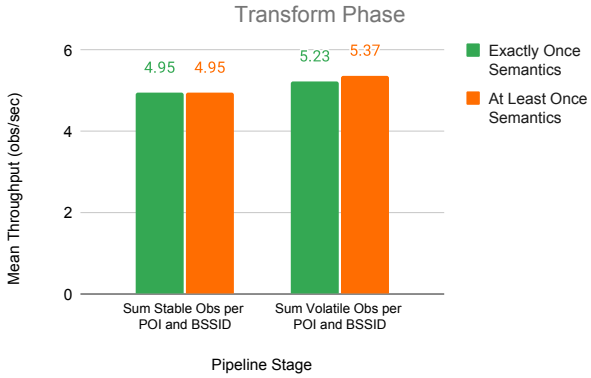

Fig. 11: Mean throughput per pipeline stage in the transform phase for each semantics.

two peaks at 2 and 7 minutes in the *exactly once* semantics pipeline, both are performing at a similar level, as plotted in Figure 13. The differences observed total to a speedup of the *at least once* semantics pipeline over the *exactly once* semantics pipeline of $1.08x$ and $1.05x$ for the mean and median, respectively.

3) Summary: In summary, although the impact of using *exactly once* semantics is significant on the achievable throughput, this impact is attenuated by the sum per key stage of the pipeline (combine function) which is naturally present in both pipelines, regardless of their processing semantics. Additionally, the main goal of our stream pipeline is to provide low-latency updates with the highest level of correctness, thus based on the median data watermark lag speedup of $1.05x$, we conclude that the ability to provide correct updates, by ensuring *exactly once* semantics, for this specific workload out weights the minor performance impact.

VI. Conclusion and Future Work

In this paper, we presented CROSS City Cloud, a cloud-native location certification system for consumer mobile applications, capable of producing and validating time-bound lo-

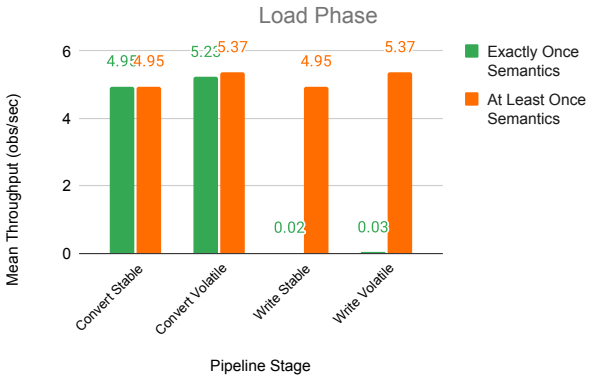

Fig. 12: Mean throughput per pipeline stage in the load phase for each semantics.

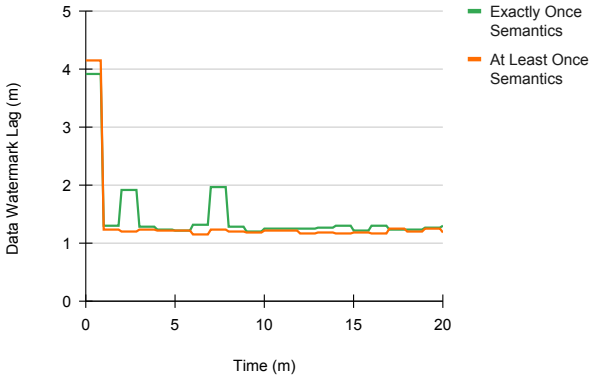

Fig. 13: Data watermark lag per pipeline processing semantics.

cation proofs. We used a smart tourism application as testbed, and demonstrated the feasibility of a Location-Certification-as-a-Service (LCaaS) platform embedded in cloud computing environments.

CROSS City Cloud ingests, aggregates and integrates scavenged network observations in intermediate network sets, which are subsequently used to fetch the stable and volatile networks of a particular point of interest, both offline and in real-time, producing and validating time-bound location proofs solely using the publicly available Wi-Fi network infrastructure. Our contribution leveraged public cloud computing technology to deploy the system, including an additional control plane to ease service operation. The evaluation stressed each layer of the system in various aspects, such as performance and scalability, through various real-world scenario assessments, and validated our solution for the expected use case. Stable and volatile set match success rates of 61.11% and 63.89%, respectively, demonstrate location and time-bound proof feasiblity. Scalability and performance analysis demonstrated that the system is able to scale horizontally, maintaining the acceptable performance level of under 100 ms under load with up to 50 concurrent users, at a 60% resource utilization. Additionally, the pipeline solution is able to provide real-time low-latency updates while enforcing a greater level of correctness.

In future work, we can explore additional window type and size parameter adjustments tailored to lower latency application use cases, such as transportation. The business logic lifecycle of CROSS City can be automated. Moreover, a system operator dashboard, as a single user interface used to operate CROSS City Cloud, with the goal of offering both health status information and query capabilities could further improve the system operator user experience (UX) of the LCaaS platform..

VII. Acknowledgements

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 (INESC-ID) and through project with reference PTDC/CCI-COM/31440/2017 (SureThing).

References

[1] J. H. Lee and R. M. Buehrer, "Location spoofing attack detection in wireless networks," in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. IEEE, 2010, pp. 1–6.

[2] Z. Zhu and G. Cao, "Applaus: A privacy-preserving location proof updating system for location-based services," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 1889–1897.

[3] E. S. Canlar, M. Conti, B. Crispo, and R. Di Pietro, "Crepuscolo: A collusion resistant privacy preserving location verification system," in *2013 International Conference on Risks and Security of Internet and Systems (CRiSIS)*. IEEE, 2013, pp. 1–9.

[4] X. Wang, A. Pande, J. Zhu, and P. Mohapatra, "Stamp: Enabling privacy-preserving location proofs for mobile users," *IEEE/ACM transactions on networking*, vol. 24, no. 6, pp. 3276–3289, 2016.

[5] G. A. Maia, R. L. Claro, and M. L. Pardal, "CROSS City: Wi-Fi Location Proofs for Smart Tourism," in *International Conference on Ad-Hoc Networks and Wireless*. Springer, 2020, pp. 241–253.

[6] R. Claro, S. Eisa, and M. L. Pardal, "Lisbon hotspots: Wi-fi access point dataset for time-bound location proofs," *arXiv preprint arXiv:2208.04741*, 2022.

[7] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, "Totp: Time-based one-time password algorithm," Tech. Rep., 2011.

[8] N. Marz, "How To Beat The CAP Theorem," http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html, Thoughts from the Red Planet, 2011, Accessed: 01-12-2021.

[9] J. Warren and N. Marz, *Big Data: Principles and best practices of scalable realtime data systems*. Simon and Schuster, 2015.

[10] J. Lin, "The Lambda and the Kappa," *IEEE Internet Computing*, vol. 21, no. 05, pp. 60–66, 2017.

[11] J. Kreps, "Questioning the Lambda Architecture," https://www.oreilly.com/radar/questioning-the-lambda-architecture/, O'Reilly.com, 2014, Accessed: 01-12-2021.

[12] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. "O'Reilly Media, Inc.", 2017.

[13] N. J. Gunther, *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Berlin, Heidelberg: Springer-Verlag, 2006.

[14] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485. [Online]. Available: https://doi.org/10.1145/1465482.1465560