LoCaaS: Location-Certification-as-a-Service

Lucas H. Vicente, Samih Eisa, Miguel L. Pardal
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal
{lucasvicente, miguel.pardal}@tecnico.ulisboa.pt, samih.eisa@inesc-id.pt

***Abstract*—Millions of tourists each year use smartphone applications to discover points of interest. Despite relying heavily on location sensing, most of them are susceptible to location spoofing, but not all. *CROSS City* is a smart tourism application that rewards users for completing tourist itineraries and uses location certificates to prevent attacks. In this case, the location verification relies on the periodic collection of public Wi-Fi network observations by multiple users to make sure the travelers actually went to the points of interest.**

In this paper, we introduce the Location-Certification-as-a-Service (LoCaaS) approach, supported by a cloud-native and improved location certification system, capable of producing and validating time-bound location proofs using network data collected by tourists' mobile devices. We show that the system can efficiently compute the stable and transient networks for a given location that are used, respectively, to validate the location of a tourist and to prove the time-of-visit. The system was deployed to the Google Cloud Platform and was validated with performance experiments and a real-world deployment.

***Index Terms*—Location Spoofing, Location Certificate, Security, Cloud Computing.**

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*, *weather services*, *location-based games*, and *smart tourism*. 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 certification 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.

CROSS City [5] is a mobile application, developed for a smart tourism use case, that utilizes location certificates. Tourists use their smartphones to interact with existing infrastructure at points of interest in the city, periodically collecting data about them. In the end, when a tour is completed, rewards are awarded for completing it. However, rewards also entice bad actors to illegitimately obtain them. To combat this, CROSS relies on a combination of strategies for producing location proofs, namely: scavenging of Wi-Fi identifiers, one-time codes broadcast by Wi-Fi beacons, and user interaction with kiosks. To reduce the need for the infrastructure approaches, which are more expensive with beacons and kiosks, the scavenging strategy [6] can be improved with more frequent and timely data processing. This is where a new and optimized cloud offering can make a difference. We call this approach Location-Certification-as-a-Service (LoCaaS). By leveraging the cloud, we can also improve the reliability, availability, scalability of the system and make it universally accessible, enabling location verification for many more consumer applications.

In this paper we implement the 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 LoCaaS capabilities in a public cloud.

II. Background and Related Work

Maia et al. [5] originally proposed CROSS (loCation pROof techniqueS for consumer mobile applicationS) providing 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 infrastructure using their mobile devices, recording traces of information. Three logical entities are defined: *prover*, that makes a claim with location evidence, *witness*, which endorses a claim with their own collected evidence, and *verifier*, that analyzes evidence and makes the decision to issue a location certificate. CROSS used a client-server model consisting of a mobile application and a centralized server with a database component. The server handles the validation of the location evidence submitted by the tourists. CROSS employs three strategies for location verification, with different trade-offs of security versus infrastructure needs: *scavenging*, with user-collected Wi-Fi traces compared against the list of known networks at that location; *TOTP*, which includes in the broadcast network SSID a time-based One-time Password (OTP) similar to the one proposed in RFC 6238 [7] but requires the deployment of a customized Wi-Fi AP (Access Point); and *kiosk*, which requires interaction with a trusted device, such as a ticket booth, for example.

Claro et al. [6] studied the scavenging approach in detail and collected a dataset of Lisbon *hotspots*. They also developed data structures and algorithms to determine the location and time interval of a visit leveraging diverse ad-hoc witnesses, to observe *long-lived* and *short-lived* hotspots, and use reported co-locations to prove the location or time-of-visit, respectively. A prover's location claim uses collected Wi-Fi AP SSIDs (*observations*) as evidence. The model also defines time windows to bound the set of observations used for verification: *epoch*, which is the most encompassing time frame, where 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*, which is a subdivision of

an epoch, where only observations collected within this time window will be used to compute the volatility of the Wi-Fi networks at a given location; and *span*, which is a subdivision of a period where prover and witness must share observations. More formally, a span is the interval defined by the claimed time-of-visit (t_p) and an additional parameter *delta* (δ) between $t_p - \delta$ and $t_p + \delta$ where evidence of co-location could be found.

For the LoCaaS cloud, we considered alternative architectures: Lambda and Kappa. Both are suited to data processing systems that require real-time data analytics. These systems require asynchronous data transformations with minimal delay without sacrificing the processing of historical data.

The *Lambda architecture* [8] has the goal of achieving both real-time and historical data processing capabilities by combining both batch and stream methods. The Lambda architecture is composed of three distinct layers: *batch*, *speed*, and *serving*. 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. The speed layer is introduced to compensate for high latency updates of the batch layer and it incrementally computes a series of views of recent data. Queries are handled by the serving layer against the merged results of both the batch and realtime views [9]. However, this process makes Lambda complex, because two separate processing systems need to be indefinitely maintained and synchronized to correctly integrate the missing updates that occur during a batch job [10].

The *Kappa architecture* [11] was designed to overcome the limitations of Lambda. In Kappa, there is no notion of batch, every data is treated as a stream and therefore only a stream processing engine is required. It consists of two distinct layers: *stream* and *serving*. Data is fed 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. Data may still be reprocessed by streaming through historical data. Kappa achieves a general-purpose solution with both real-time and reprocessing capabilities without the added complexity and maintenance needs of two separate subsystems.

III. Architecture

The CROSS client application fetches the catalog of itineraries, logs the visits to each point of interest (sensed Wi-Fi signals), and either stores data locally and sends it later or immediately publishes it to the server. 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. Solely with offline or batch processing, any location proof requests made would necessarily have to be purposely delayed until the end of the current period, or multiple high latency batch jobs would have to be triggered during short periods. To avoid this, we opted for an extended Kappa architecture with three distinct layers: *domain*, *stream*, and *serving*. The domain layer is added to store and handle 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 observation data as atomic and immutable facts, kept for the epoch, with publish timestamps to allow for recomputation of historical views. Additionally, the stream layer produces 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 1 illustrates the resulting CROSS City server-side architecture.

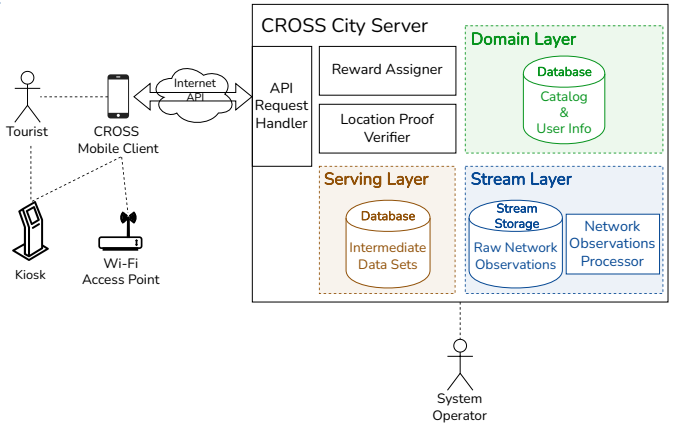

Fig. 1: Overview of the server-side architecture.

A. Collection of Network Observation Data

Observations collected by tourists must be continuously integrated into the operational dataset. The data lifecycle is illustrated in Figure 2 and it 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 this stage, only the trusted system operators submit network observations of each existing point of interest, with the goal of deriving the initial stable network sets. The *Live* stage is a sequence of epochs, with the initial one named $epoch_1$. Throughout this stage, untrusted entities interact through trip submissions. Each visit to a location contains evidence (network observations) which is 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 confidence threshold of the claimed location is fulfilled, does the visit get accepted and its network observations are integrated in subsequent stable and volatile network sets.

B. Computation of Intermediate Observation Set

The main goal of the stream layer is to produce network observation views to be queried efficiently, in a low-latency manner. Hence, an *incremental computation* approach was adopted over *recomputation*, avoiding the execution of function logic over the entire set of observations. To be efficient, the views should contain intermediate results of the expected queries: *stable networks set* (most observed networks, over an

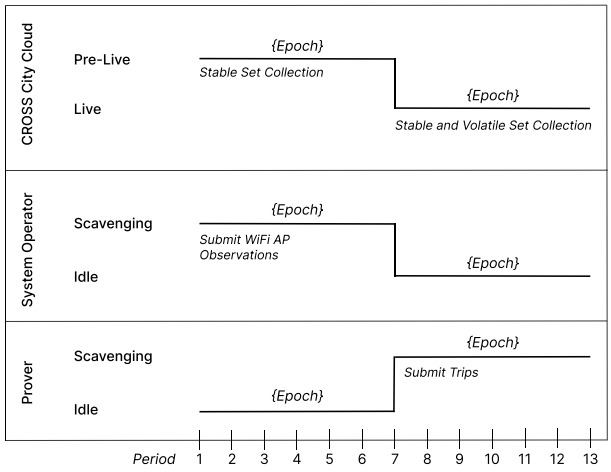

Fig. 2: Timing diagram for the CROSS City data lifecycle across $epoch_0$ and $epoch_1$.

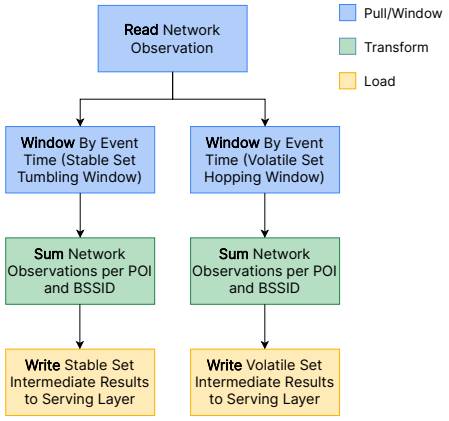

Fig. 3: Processing pipeline for producing intermediate stable and volatile network sets.

epoch, for a given point of interest) and *volatile networks set* (least observed networks in a span interval, over a period, for a given point of interest).

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. Stable network sets are queried within an epoch with a minimum granularity of a *period*. Volatile network sets are queried within a span, and each of the possible span time windows encompasses smaller intervals of size equal to the greatest common divisor of the span set. 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 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 3 represents the pipeline, where 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 span set, then keyed and summed per point of interest and network identifier (BSSID), and finally written to the serving layer.

C. 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 $t\%$ observed networks over that epoch, off the critical path. Network observations from past epochs 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 $t\%$ observed networks of the previous epoch and retrieve the bottom $t\%$ observed networks within the claimed time interval. In the implementation, we defined t=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 4 details the deployed cloud architecture on the Google Cloud Platform (GCP).

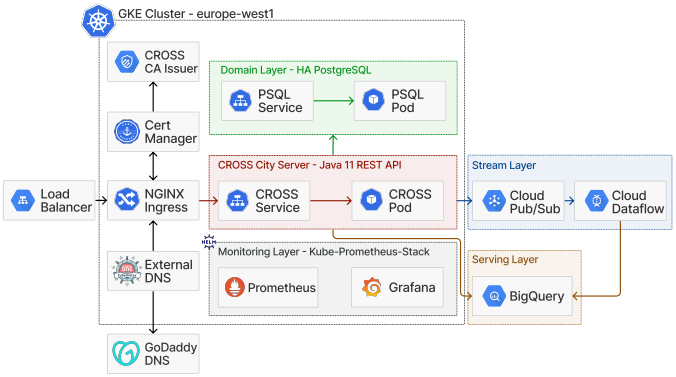

Fig. 4: Overview of architecture deployed to the Google Cloud.

A. Domain Layer

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

The *database* component of the domain layer stores both user and tourism related data. We maintained the use of a relational database, namely, PostgreSQL (HA PostgreSQL in Figure 4).

The *compute* (CROSS City Server in Figure 4) is the entry-point for the remote invocations, and manages user sessions, handles trip submissions and the respective location claim verification for reward attribution. The interface is implemented in Java and follows the REST (REpresentational State Transfer) style for web services. The client communicates with the API server over HTTPS exchanging payloads defined and encoded with protocol buffers[1].

To mitigate cloud provider lock-in, we deployed both components to a Kubernetes-based cloud service, specifically to Google Kubernetes Engine (GKE), ensuring control over the container orchestration, including networking, storage, and observability of each component.

B. Stream Layer

The stream layer is comprised of two primitive components: data ingestion and data processing. The *data ingestion* consumes observation events, published by clients through the API, for streaming into the data processing component. This component is a message broker to avoid direct communication between producers and consumers. 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 equal to 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. The Google Cloud Pub/Sub component was selected, as it fulfills the stated requirements (Cloud Pub/Sub in Figure 4).

The *data processing* component (the pipeline) processes the network observations to produce the intermediate results for the stable and volatile set queries. The pipeline aggregates network observations on event-time in two separate tumbling and hopping windows - corresponding to the stable and volatile set windows. 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. The Apache Beam[2] open-source framework for parallel, distributed data processing at scale was selected for use. It can plug into the Google Cloud Dataflow service to fulfills our needs (Cloud Dataflow in Figure 4). Dataflow assures *at least once* semantics, by default. However, 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 unique IDs and specific functions provided by the services used.

[1]https://developers.google.com/protocol-buffers

[2]https://beam.apache.org/

C. Serving Layer

The serving layer is composed of a single primitive component, the database, that persists the aggregate intermediate results computed by the stream layer and serves query requests related to the stable and volatile 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. To guarantee proper inter-layer operability and connectivity the database should be easily integrated with both the data processing stream layer component (Cloud Dataflow) and the API domain layer component (Kubernetes pod with the Java REST server). Writes are expected to be made in real-time, so the database component must support streaming records to it, and reads may be performed in random order. Additionally, the database must scale as the size of the intermediate results increases and still 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 opted for 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 a function with *exactly once* semantics (BigQuery in Figure 4).

V. Evaluation

We split the evaluation in two parts: the location certification feasibility and performance assessment.

A. Experimental Setup

The selected benchmark tool and CROSS City Cloud were packaged as Docker containers and deployed to distinct Google Kubernetes Engine (GKE) clusters, comprised of one and two nodes, respectively, physically isolated, in the same cloud region. A comprehensive specification of the GKE clusters used is shown in Table I.

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

To test the feasibility of our solution in providing location and time-bound proofs, we used the *LXspots* dataset [6] with network observation data collected in the city of Lisbon, Portugal. Each point of interest has touristic relevance and different characteristics, such as being outdoors or indoors, sparse or central, and central or remote. The locations were: Alvalade, Comércio, Gulbenkian, Jerónimos, Oceanário, and Sé. Each smartphone used for data collection - Samsung Galaxy S9, Huawei Mate 10 and LG V10 thinq - represents a distinct prover, called Alice, Bob and Charlie. Each prover stays at a location for 15 minutes. We used the dataset portions of 7 consecutive days as *epoch*, from 2019-07-29 to 2019-08-04, and 1 day as *period*, namely 2019-08-19.

TABLE I: Kubernetes 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

1) Stable Set Match as Location Proof: To prove presence at the location, the collected network observation set of the prover 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 need a lower match threshold or the deployment of known TOTP Wi-Fi APs.

2) Volatile Set Match as Time Proof: To prove the visiting period, the scavenged network observation set of the prover 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 in at least 50% of the claimed span intervals. Most notably, 75% 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%, which shows an effective attesting of the visiting period.

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%	
Comércio	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%	
Jerónimos	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%	
Oceanário	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%	
Sé	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%	

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 focus on the trip submission through the domain layer. The integration of scavenged network observations through the stream and serving layer is done asynchronously and will be evaluated separately in Section V-D. 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 points 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 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 more server pods (*replicas*) to accommodate a growing demand. During the execution of the workload, the request response time, rate of requests and both the CPU and memory usage metrics were collected using the k6 open-source load testing tool.

1) System Performance and Scalability: Using the throughput measurements collected per level of user concurrency, we are able to model the scalability of the system with the *Universal Scalability Law* (USL), shown in Equation 1, where X stands for throughput, N for concurrent users, λ for performance coefficient, σ for serial portion, and κ for crosstalk factor. The USL [12] extends *Amdahl's law* [13] (shown in Equation 2) with an additional parameter (κ), allowing us to model capacity degradation related to coherency losses. The USL 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

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

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.

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

$$Amdahl's\ law : X(N) = \frac{\lambda N}{1 + \sigma(N-1)} \tag{2}$$

Table III summarizes the resultant performance coefficient and scalability parameters estimations for each configuration, with their respective model plots in Figure 5. 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 (Theorem 4.3 Maximum Capacity [12]) 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 2.77 times.

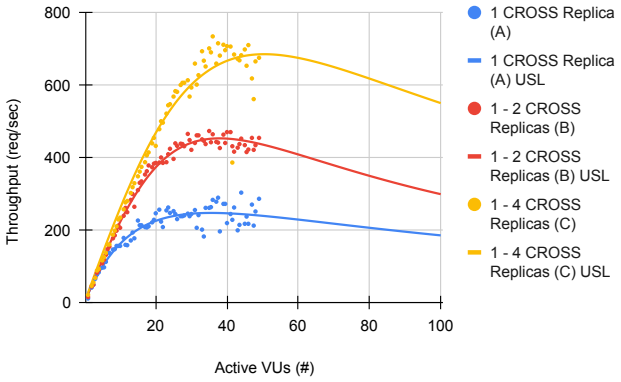

Fig. 5: Throughput over Active Concurrent Virtual Users for each system size configuration and respective USL model.

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

2) Request Performance: From the collected mean latency measurements plotted in Figure 6, we note that for a latency threshold of 100 ms (the limit for giving the user the perception that the system is reacting instantly [14], [15]), both configurations *A* and *B* are only able to perform below set threshold solely until 17 and 37 concurrent virtual users, respectively. 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. To further quantify the user perception of the system, we plotted the percentile-90 latency (filters top 10% worse latencies) in Figure 7. Percentiles are useful to determine the expected maximum response time for a percentage of requests/users. In this case, only configuration *C* is able to withstand a set latency threshold of 200 ms (double the set mean latency threshold), i.e., 90% of users will experience a response time either as fast or faster than 200 ms.

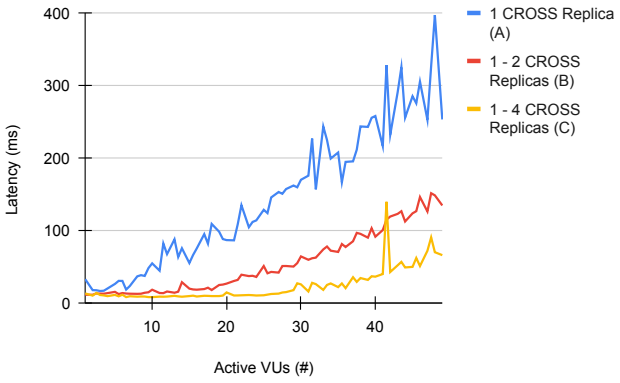

Fig. 6: Mean Latency over Active Concurrent Virtual Users.

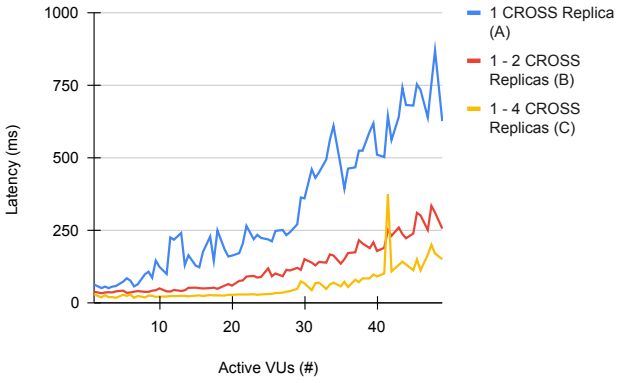

Fig. 7: Percentile 90 Latency over Active Concurrent Virtual Users.

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.

3) Discussion: For this specific workload, configuration *C* outperforms the other two in both system and request performance metrics, as observed and predicted. Moreover, we can infer that the system scales horizontally, maintaining an acceptable level of performance and resource utilization.

D. Stream Layer Performance

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 as the collected Wi-Fi AP evidence set of a tourist's visit to a given point of interest, 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 network observation with the latest publish time (watermark) has been processed and outputted.

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

1) Impact on Throughput: In Figure 8, the plot shows a speedup, of approximately 3.5 times, with *at least once* semantics all throughout the pull/window phase (Figure 3) in which each stage performs consistently at the same rate. Nonetheless, as noted in the transform phase (Figure 3) plot in Figure 9 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 1.0 times speedup), indicating a potential bottleneck at these stages. In the load phase (Figure 3) plotted in Figure 10, with *at least once* semantics the pipeline is able to sustain the rate of observations from the previous phase, 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 (deduction of 165 times speedup).

2) Impact on Output Data Watermark: The data watermark lag allows us to quantify the processing time in relation to the publish time of an observation. 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, as plotted in Figure 11.

3) Discussion: The impact of using *exactly once* semantics is significant on the achievable throughput, but this impact is attenuated by the *sum per key* stage of the pipeline, which is the combine function responsible for grouping and counting the observations of the networks with the same SSID at

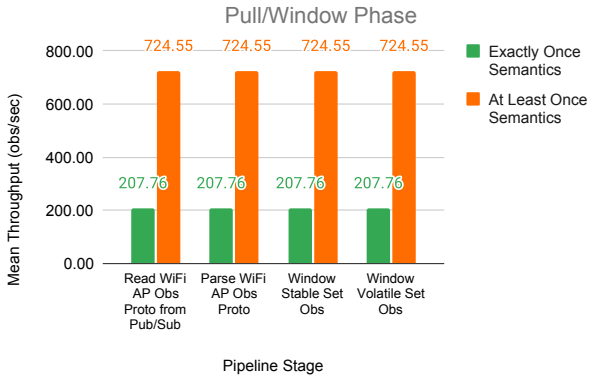

Fig. 8: Mean throughput per pipeline stage in the pull/window phase for each semantics.

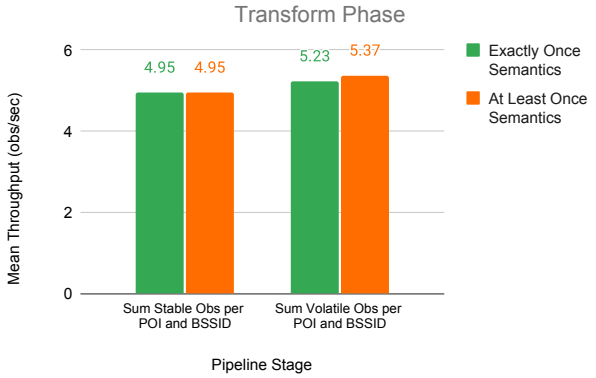

Fig. 9: Mean throughput per pipeline stage in the transform phase for each semantics.

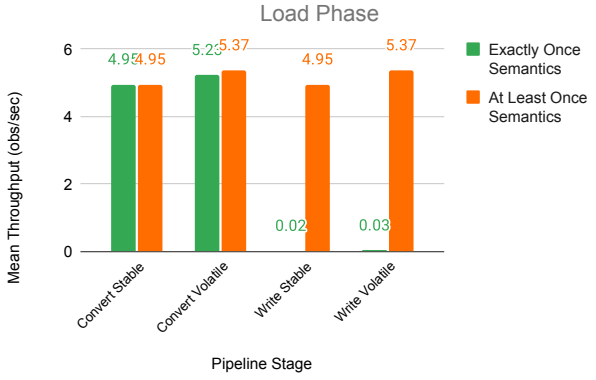

Fig. 10: Mean throughput per pipeline stage in the load phase for each semantics.

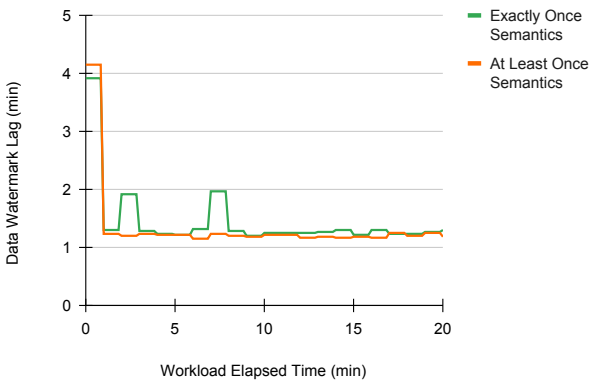

Fig. 11: Data watermark lag per pipeline processing semantics.

each point of interest, and naturally present in both pipelines regardless of their processing semantics. Additionally, the main goal of our stream pipeline is to provide correct stable and volatile network set view updates at a low-latency. With *exactly once* semantics, we are able to ensure the highest level of correctness. Moreover, the median data watermark lag speedup of 1.05 times demonstrates the minor performance impact of enforcing *exactly once* semantics over *at least once* semantics. As a result, we conclude that the ability to provide correct updates outweighs the performance impact for this specific workload, allowing us to meet the main goal.

VI. Conclusion

In this paper, we presented a cloud-native location certification system for consumer applications, capable of producing and validating time-bound location proofs. We used CROSS City, a smart tourism application as testbed, and demonstrated the feasibility of a Location-Certification-as-a-Service (LoCaaS) platform in a cloud computing environments. With the provided services, CROSS is able to better ingest, aggregate and integrate scavenged network observations in intermediate network sets, to compute the stable and volatile networks of a particular point of interest. Our contribution leveraged public cloud computing technology to deploy the system. The evaluation stressed each layer of the system in regards to performance and scalability, through various scenario assessments, and validated the solution for the expected use case. Stable and volatile set match success rates of 61.11% and 63.89%, respectively, demonstrate the feasibility of using the computed sets to validate the location and time claims. 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 low-latency updates while still enforcing *exactly once* processing semantics, which is important for the location assurance algorithms.

In future work, we plan to provide an *operator dashboard*, for the LoCaaS platform as a novel cloud offering that can be used by future applications to achieve certified location and thwart many attacks.

VII. Acknowledgements

We thank Prof. Nuno Santos and his team for their cloud support and funding. We also express heartfelt appreciation to Rui Claro for gathering Wi-Fi data around Lisbon.

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," Internet Requests for Comments, RFC Editor, RFC 6238, 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] N. J. Gunther, *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Berlin, Heidelberg: Springer-Verlag, 2006.

[13] 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

[14] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, 1968, pp. 267–277.

[15] S. K. Card, G. G. Robertson, and J. D. Mackinlay, "The information visualizer, an information workspace," in *Proceedings of the SIGCHI Conference on Human factors in computing systems*, 1991, pp. 181–186.