A fog-based distributed look-up service for intelligent transportation systems

G. Tanganelli, C. Vallati, E. Mingozzi
Dip. Ingegneria dell'Informazione, University of Pisa
L.go L. Lazzarino 1, I-56122
Pisa, Italy
{g.tanganelli, c.vallati, e.mingozzi}@iet.unipi.it

***Abstract*—Future intelligent transportation systems and applications are expected to greatly benefit from the integration with a cloud computing infrastructure for service reliability and efficiency. More recently, fog computing has been proposed as a new computing paradigm to support low-latency and location-aware services by moving the execution of application logic on devices at the edge of the network in proximity of the physical systems, e.g. in the roadside infrastructure or directly in the connected vehicles. Such distributed runtime environment can support low-latency communication with sensors and actuators thus allowing functions such as real-time monitoring and remote control, e.g. for remote telemetry of public transport vehicles or remote control under emergency situations, respectively. These applications will require support for some basic functionalities from the runtime. Among them, discovery of sensors and actuators will be a significant challenge considering the large variety of sensors and actuators and their mobility. In this paper, a discovery service specifically tailored for fog computing platforms with mobile nodes is proposed. Instead of adopting a centralized approach, we propose an approach based on a distributed hash table to be implemented by fog nodes, exploiting their storage and computation capabilities. The proposed approach supports by design multiple attributes and range queries. A prototype of the proposed service has been implemented and evaluated experimentally.**

Keywords—Fog Computing; Distributed Hash Table; Look-up service; Intelligent Transportation Systems;

I. INTRODUCTION

Future Intelligent Transportation Systems (ITS) will greatly rely on sensors and actuators deployed on vehicles and the roadside. Such sensing and actuating infrastructure will enable new applications [6] that will improve many aspects of transportation such as (i) safety, e.g. through prompt distribution of road hazard warnings to drivers; (ii) traffic efficiency, e.g. through timed detection and signaling of traffic jams; and (iii) drivers' awareness, e.g. through the dissemination of automatic map updates and info. The implementation of these services is expected to largely rely on cloud computing as a scalable and cheap computing and storage infrastructure [7]. However, for services having low-latency requirements and demanding location-aware interactions, running in data centers usually located far from devices and users may not be a viable option.

To this aim, fog computing [13] has recently emerged as a new computing model that delivers services and storage at the edge of the network close to physical systems and users. Fog computing is expected to play a major role for applications and services deployed on devices installed in the roadside infrastructure or directly on connected vehicles. Such runtime environment will provide a virtualized framework to support applications, such as real-time traffic accident notification or localized information delivery [8], which are unfit for the cloud environment.

Among the basic services that need be delivered to (ITS) applications in this context, maintaining a directory of the available resources, i.e., sensor and actuator devices, for efficient discovery is of utmost importance. Implementing such a service in a cloud infrastructure is rather straightforward, since centralized solutions can be readily adopted. However, a fog environment is inherently dispersed, and therefore providing a distributed implementation for the discovery service represents a better fit. Moreover, resource discovery for ITS applications presents additional challenges. On the one hand, resources will be characterized by different and heterogeneous attributes. Handling such variety will be further challenging given the dynamicity of the environment in which context data is created and destroyed rapidly, e.g., because of mobility. On the other hand, applications will specify queries with both multiple attributes and values in a range. For example, consider charging stations for electric vehicles (EVs), they might be equipped with sensors to check if a charging spot is available. In this case, an EV might want to discover all the charging stations of a certain type in its proximity, i.e. within a certain range of latitude and longitude. In this case, the application of the EV will issue a discovery request with multiple attributes (type, latitude, and longitude) and with range-query (latitude and longitude within a range).

In this paper, a distributed discovery service for fog computing platforms is presented. The proposed look-up service exploits a two-tier logical overlay, inspired by LORM [4], an existing Distributed Hash Table (DHT), to support both *multi-attribute* and *range* queries, as required by ITS applications. In the proposed service, the original design is extended to allow dynamic management of information, required to handle the heterogeneous storage capabilities offered by fog nodes. Specifically, the proposed service can handle dynamic creation of information contexts that could vary over time due to mobility. The feasibility of the proposed approach is demonstrated through a prototype based on existing software, which is exploited to run a performance evaluation.

The rest of the paper is organized as follows: in Section II an overview of the state of art on the topic of distributed services for discovery is provided. Section III presents the overall architecture of the look-up service while in Section IV its basic operations are presented, Section V presents the performance evaluation, while in Section VI conclusions are drawn.

II. State of Art

Many distributed discovery services based on DHT have been proposed in the literature. However, among them, only a few allows multi-attribute and range queries.

In [1], the authors present MAAN, a Multi-Attribute Addressable Network, derived from Chord [2], which is specifically designed for the grid computing paradigm that requires multi attribute queries. Every node has a unique ID and all nodes form an overlay that has a ring topology based on their IDs. In order to allow range queries, MAAN uses a *locality preserving* hash function. Such hash function simplifies the resolution of range queries: starting from the lower bound of the query range, nodes can be interrogated in sequence and, whenever a value greater than the request's upper bound is obtained, a result is found. MAAN also allows multi attribute queries by registering multiple pairs (*key*, *value*) for each resource; the multi attribute lookup is split into sub-queries, executed in parallel and reassembled at the request originator. However, this approach increases the exchange of information and, thus, the overhead.

Mercury [3], instead, allows multi-attribute queries by exploiting multiple DHT overlays. Each DHT manages a single attribute. In order to enable range-queries, keys are strictly ordered. The proposed architecture, however, is not efficient: to manage multiple separate DHTs is expensive, especially when a large number of attributes is considered. Considering that nodes participate to every DHT, the number of routing tables stored by each node is in the order of $m \log(n)$, where n is the number of nodes, and m is the number of attributes.

The limitations of MAAN and Mercury are fixed by LORM, proposed in [5]. Lorm is a DHT system designed to provide scalable multi-attribute and range-query resource discovery. In order to support storage and retrieval of data associated with multiple attributes, Lorm adopts an overlay network in which nodes are grouped into clusters, each one managing the information related with a different attribute. In order to enable retrieval of information from different clusters, a two-dimensional hash schema is exploited: one dimension is used to identify the cluster that manages the attribute, the other one to locate the information inside the cluster. In order to handle range queries, a locality preserving function is adopted to compute the part of the index that is used to retrieve the information inside the cluster. In order to guarantee efficient routing across clusters, a specific routing strategy is defined, in which the routing tables include also neighbors from different clusters. This overlay architecture that splits information into clusters jointly with the routing strategy minimizes the number of hops for information retrieval compared to MAAN and Mercury.

Although LORM is efficient and offers both multi-attribute and range query functionalities, it has some limitations that refrain from its adoption in ITS. One limitation is represented by the number of attributes that can be managed. Specifically, the number of attributes must be fixed and known at the time of configuration. The addition of a new attribute through the addition of a new cluster is not addressed, and would require significant modifications to the current design and, in particular, the hash schema. Another limitation is the distribution of the information load thought the nodes of the overlay. Although LORM guarantees a more balanced distribution of information among the nodes, the hash function does not take into account potentially heterogeneous storage capabilities between nodes. In order to solve both these issues, in this paper we present a DHT that adopts an architecture, inspired by LORM, but can handle dynamic creation/destruction of clusters and potentially offer mechanisms for data relocation.

III. DHT Lookup Service

The proposed lookup service is implemented using a hierarchical two-tier DHT overlay. Such hierarchical structure, inspired by the architecture of LORM [4], is adopted to achieve efficient storage and lookup of data associated with multiple attributes. The overall architecture is illustrated in Figure 1. Specifically, the outer tier is composed of a set of independent DHTs overlays (called *clusters),* each one instantiated to store the resources associated with one specific attribute. Lookup requests involving a specific attribute are processed by the respective cluster. The inner tier, called *global DHT,* is exploited to interconnect the clusters of the outer tier together. Specifically, the global DHT is exploited to dispatch multi-attribute queries to different clusters. For each multi-attribute query, multiple subqueries, one for each attribute, are generated and forwarded to the corresponding cluster for resolution. The results obtained by each cluster are then aggregated and forwarded to the application.

In order to allow range queries for attributes involving numeric values, e.g. GPS location, a *locality preserving* hash function is implemented by clusters. A locality preserving hash function is a hash function in which the numerical order of the values is preserved in the hash domain, formally:

DEFINITION. A Hash function is locality preserving, or *LPH* for short, if it has the following properties: if $v_i < v_j$, then $LPH(v_i) < LPH(v_j)$; and, if an interval $[v_i , v_j]$ is split into $[v_i , v_k]$ and $[v_k , v_j]$, respectively, the corresponding interval $[LPH(v_i) , LPH(v_j)]$ must be split into $[LPH(v_i) , LPH(v_k)]$ and $[LPH(v_k) , LPH(v_j)]$.

Such property of the hash function simplifies the lookup of range values: starting from the lower bound of the request, the nodes of the DHT can be queried in sequence; whenever a value greater than the upper bound is obtained, all the desired results are also obtained. Similarly to LORM [4], we define the locality preserving hash function as:

$$LPH = (\pi - \pi_{min}) \frac{(d-1)}{(\pi_{max} - \pi_{min})},$$

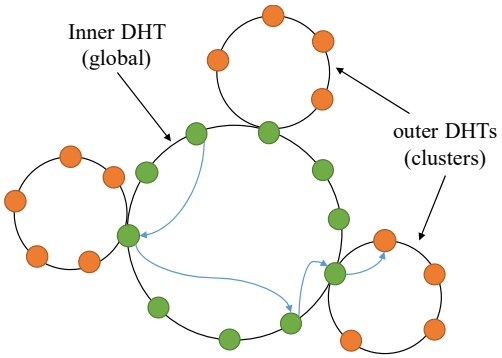

Figure 1: Two-tier Architecture

where π is the attribute value in the range $[\pi_{max}, \pi_{min}]$ and d is the max allowed value in the hash domain. In the global DHT and in the clusters that do not manage attributes with numeric values, a standard hash function, e.g. SHA-1 [11], can be adopted, hereafter H for short. It is worth to note that the proposed architecture is general enough to be realized using any DHT implementation. In our prototype, for instance, we exploited Pastry for both the clusters and the global DHT.

The overlay architecture shown in Figure 1 is a logical architecture that allows efficient storage and lookup of multiple attributes. Its implementation exploiting actual Fog devices, however, must take into account also other requirements. Primarily, the implementation should consider that Fog nodes are physical devices characterized by heterogeneous computation and storage capabilities [12]. For this reason, in order to obtain an efficient distribution of data, each Fog node should join a number of clusters proportional to its capabilities. Secondly, the implementation should support the dynamic management of attributes. Considering that the set of attributes might change over time, the implementation is required to support the introduction (deletion) of new (existing) attributes efficiently.

In order to support such requirements, the implementation of the logical architecture has been designed to allow Fog devices to host multiple DHT nodes at the same time. The DHT nodes to be hosted could be selected based on applications' preference, for example an application that often retrieves resources based on their location might want to manage Latitude and Longitude attributes. The overall structure of a Fog device is illustrated in Figure 2. The Fog device instantiates multiple DHT nodes joining different clusters, one for each attribute that the Fog device is configured to manage. Each logical node communicates with the other DHT nodes of the cluster to manage the look-up requests that involve the attribute of the cluster. Logical nodes are managed by a *Manager node* (MGN), which is responsible for

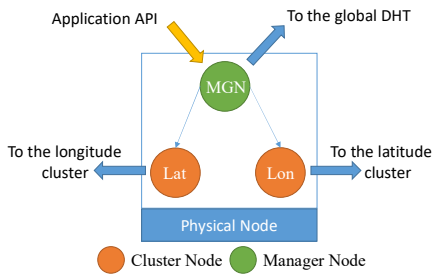

Figure 2: Node internal architecture

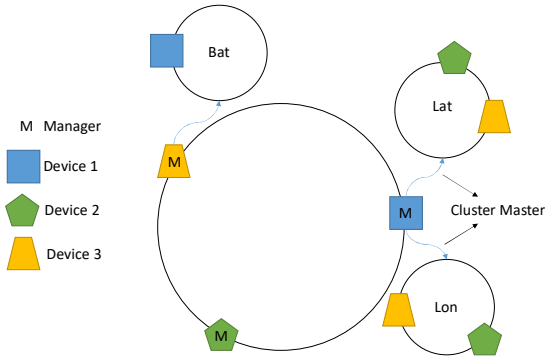

Figure 3: System Architecture

dynamically creating or destroying new or existing instances of logical nodes upon request. In addition to this, the manager is responsible for handling the dispatch of the lookup requests originated from the local applications to the proper cluster. To this aim, the manager provides an interface that allows applications executed locally on the Fog device to issue lookup requests, which are then dispatched to the proper cluster. In order to manage lookup requests involving attributes for which a local node does not exist, the manager node joins the global DHT. The global DHT contains a list of selected manager nodes, called Cluster Master nodes (CMT), that are selected as last resort to resolve lookup requests for a specific attribute. To this aim, for each attribute one CMT is selected among all the MGNs that have a connection with that cluster. Each CMT is published in the global DHT using as key its attribute, e.g. *"Battery Level"*, allowing a MGN to locate the CMT of an attribute through a simple lookup query on the attribute name. After retrieving the CMT, the MGN can forward the request, which is processed by the CMT using the local DHT node connected to the cluster of the attribute.

Let us explain further the procedure through an example. Let us suppose that a Fog device (Nx) is running a local application, which is interested in retrieving resources based on their location. In order to minimize the response delay, the Manager node (Nx-MNG) decides to manage locally the two attributes, i.e. latitude and longitude, creating two logical nodes, Nx-Lat and Nx-Long, that join the latitude and longitude clusters, respectively. Exploiting the APIs exposed by the Nx-MNG, the application can perform queries on the latitude and the longitude, which are handled locally by the Nx-MNG that forwards the requests to Nx-Lat or Nx-Long. If, instead, the application performs a query involving another attribute, e.g. battery, the Nx-MNG can interrogate the global DHT to retrieve the CMT responsible for such attribute, in this case the battery, and then forward the request. The CMT that receives the request handles the query contacting the local cluster node that manages the battery attribute and then sends back the response.

Finally, the mapping of the logical overlay into the physical architecture considered in the example is presented in Figure 3. In this example, three physical Fog devices, represented using different shapes, implement the distributed discovery service. Three different clusters are created to handle the information related with three different attributes, Latitude, Longitude and Battery, respectively. As can be seen, device1 and device2 decide to manage the Latitude and Longitude attributes, creating two

Table I: APIs

API	Description
joinGlobal()	Join to the global DHT.
insertGlobal(H(attributeName), info)	Add the information of the Cluster Master node to the global DHT.
joinCluster()	Join to a cluster.
lookupGlobal(H(attributeName))	Retrieve the Cluster Master node for the attributeName in the global DHT.
lookupCluster(LPH(attributeValue))	Retrieve the info stored in the cluster for the attributeValue
insertCluster(attributeValue, info)	Add a new value to the cluster with its associated information.
rangeLookup(LPH(minValue), LPH(maxValue))	Retrieve the info stored in the cluster, for values in the range.

logical nodes that join Latitude and Longitude clusters. Device3, instead, decides to manage only Battery Level, creating a local node that joins the corresponding cluster. Every Fog device runs a Manager node that joins the global DHT. Among them, the Manager node of device1 is selected as Master Cluster for Latitude and Longitude, while the Manager node of device3 is selected as Master Cluster for Battery.

The proposed architecture can be exploited to dynamically add attributes at runtime. Instead of having a predefined set of clusters, when an application wants to register a new attribute, which is not already available in the system, the Manager node can create a new cluster, and thus the new attribute. This opportunity is particularly suited to handle the dynamicity typical of dynamic environments, such as ITS, handling changes in the environment in a seamless manner. Although it is not evaluated in this work, detachment between physical and logical nodes can be exploited to dynamically re-distribute data at runtime. Specifically, logical nodes can be migrated along with their data from one Fog device to another, thus providing the required flexibility to handle load variations. For instance, if a device is running out of storage, it can decide to request the migration of some of its DHT nodes to other devices that, instead, are less loaded, thus balancing the distribution of data.

IV. Information Flow

In order to describe in details the operations performed by the discovery service, in this section we present a detailed description of the information flow exchanged in every operation. The description is based on the APIs exposed by each single DHT module that are summarized in Table I. It is worth to highlight that these APIs can be realized through any DHT concrete implementation.

The first operation performed by a new Fog device is the procedure required to join the discovery service. Without loss of generality, we assume that at least one device has already joined the system and its address is known to devices that wants to join. The procedure required is depicted in Figure 4. The new node (Nx) sends a join request to one of the devices that have already joined (Ny), that in turn forwards the request to the global DHT. The global DHT returns all the information regarding the nodes that are already part of the system, information exploited by the Nx to build the local routing table and configure the Manager Node (Nx-MNG).

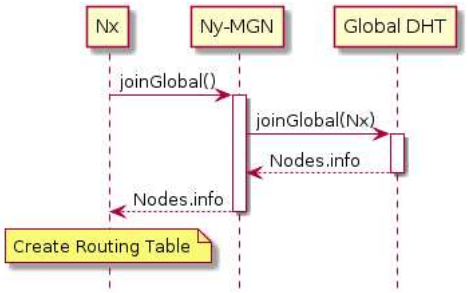

Figure 4: Manager Node creation

Subsequently, the node selects the attributes that want to manage and trigger the Manager to create the logical nodes, one for each attribute. In Figure 5 it is shown the information flow of a device Nx that wants to join the "Longitude" cluster (Lon for short). First, the Nx-MNG sends a lookup request to the global DHT for the key *"Lon"* to retrieve the address of the Cluster Master node for the Longitude attribute (CMT-Lon). Then, the Nx-MNG triggers the local DHT node to join the Longitude cluster contacting the CMT-Lon. In case the attribute is not available, the lookup to the global DHT fails. Consequently, the Nx-MNG creates a new logical node and performs the *insertGlobal(H("Lon"))* to add itself as CMT-Lon in the global DHT. From now on, the Nx-MNG will act as Cluster Master node of the Longitude attribute.

To add a new resource associated with an attribute, we must distinguish between two cases – as shown in Figure 6: i) the publishing node already has a virtual node in the cluster of the attribute, or ii) the publishing node does not have a virtual node in the cluster of the attribute. In the first case, let us suppose that Nx wants to add a new resource into the "Longitude" cluster for which it exists the Nx-Lon logical node. Nx-MNG forwards the request to its Nx-Lon node that publishes the new value in the DHT. In the second case, instead, since Nx does not have any virtual node in the "Battery Level" cluster, the Nx-MNG forwards the insert request to the Cluster Master of the Battery-Level attribute which, in turn, will insert the value into the Battery-Level cluster.

Finally, the lookup procedure is similar to the insertion reported in Figure 6; the only difference is that *insertCluster* requests are replaced by *lookupCluster* requests. When multi-attribute lookup queries are requested, they are divided into different single attribute sub-queries and resolved in parallel, thus reducing the overall response time. Specifically, each sub-query is executed in the corresponding cluster. The source node receives

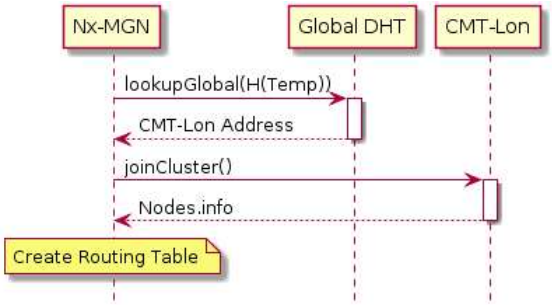

Figure 5: Cluster join

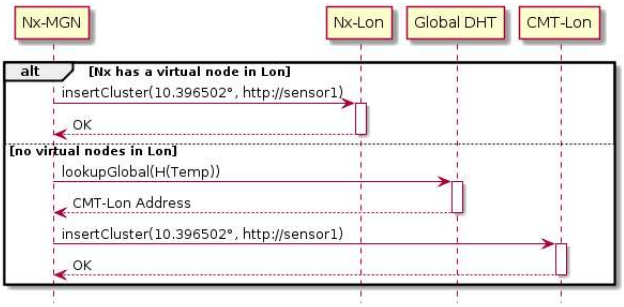

Figure 6: Insert a new value in a cluster

all the results and aggregates the resulting information that is then sent back to the application.

Range query requests are managed similarly to regular lookup requests, as the range is handled locally inside each cluster. The overall procedure is illustrated in Figure 7 by means of an example. In the example, a range query for the Latitude attribute, managed by the local *Nx-Lat* node, is issued for the range $[L_{min}, L_{max}]$. First, *Nx-Lat* issues a rangeLookup request for $[L_{min}, L_{max}]$, which is routed to the node identified by $LPH(L_{min})$, *Ny-Lat* in the example. The range lookup returns all the values stored locally by *Ny-Lat* and in addition the list of successor nodes in the cluster, if L_{max} is greater than the maximum value stored locally. Based on such information, *Nx-Lat* can send the same range lookup to all the successors, $\{N_{y+1} \dots N_{y+n}\}$, until the L_{max} value is retrieved. It is worth to note that all the range lookup requests (except the first) could be performed in parallel to reduce the overall time required to perform the range lookup operation. To this aim, the list of successors is updated using the result of each response.

V. Performance Evaluation

In order to validate the proposed solution, a prototype has been implemented. The software implements the proposed discovery architecture using Pastry. A set of experiments is then executed to assess the performance of the proposed solution. Different distributed physical devices are emulated on the same hardware using the Linux Container framework (LXC). LXC is a lightweight operating-system level virtualization framework that allows to run multiple isolated systems on a single host through the use of separate isolated containers [10]. In order to simulate the interconnection of different devices through Internet links the Dummynet software has been exploited [9].

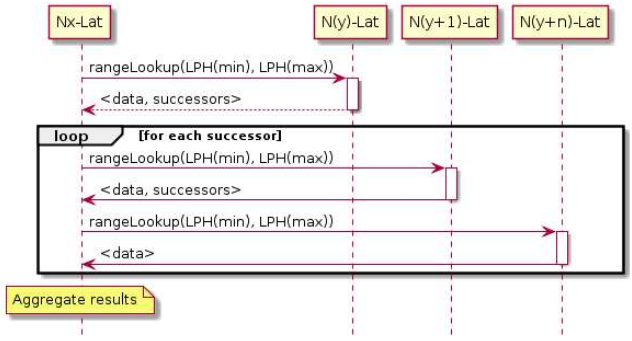

Figure 7: Range Lookup

Dummynet is a live network emulation tool that can be used to add custom delays in the stack buffers at the kernel level. Specifically each physical device is emulated through a dedicated container and, through Dummynet, we emulated a communication delay of 80ms, in order to simulate a real use case scenario. The tests are executed on an Intel Quad-core 3 GHz processor with 8 GB of RAM.

For each test the total number of managed attributes is fixed to 20; consequently 20 clusters are created, for each one 1000 resources are registered with a value of the attribute selected randomly in the range [0-100]. In all tests, requests are generated following a Poisson generation process with a rate of one request every 200ms. Each experiment is run until 100 requests are issued. In order to obtain statistically sound results, ten independent replicas for each scenarios are run, and metrics of interest are then estimated for each scenario along with a 95% confidence interval.

In the first set of experiments, we configure each node to manage a variable number C of attributes, thus creating C logical nodes, each one deployed in a specific cluster. We vary the number of physical devices to assess the response time of our architecture. It is worth to note that the overall number of logical nodes in the network is equal to $n(C+1)$, where n is the number of connected devices. Each device has C logic nodes plus the Manager Node. The performance of the system is assessed measuring the response time of a lookup operation, defined as the time between the generation of a request and the reception of the response.

In Figure 8, we show the response time of the lookup operation for a single attribute versus the number of connected devices with two different values of C, i.e $C = 5$, and $C = 10$, respectively. As can be seen, the response time grows with the number of connected devices. The increase of the response time is in line with the complexity of the retrieval of information in DHT systems, in which the number of hops increases as $O(\log n)$, with n equal to the number of logical nodes. It is worth to node that 100 devices with 10 attributes per device results in a total of 1120 logical nodes deployed in the system. On the other hand, when more attributes per device are involved, it is more likely that each device has a logical node in the requested attribute cluster: this avoids to interrogate the

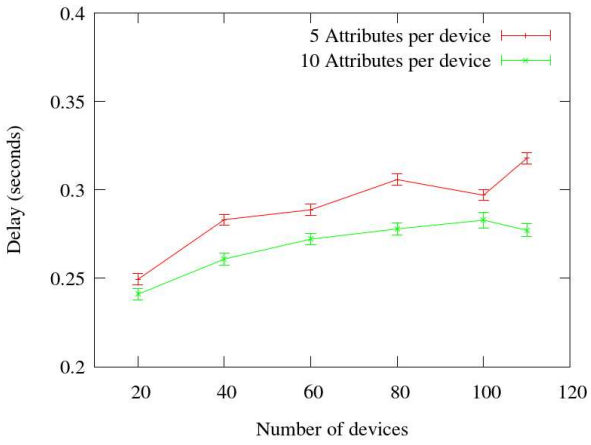

Figure 8: Response Time vs Number of nodes

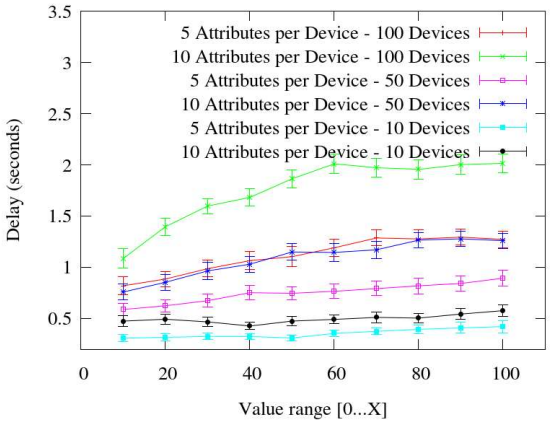

Figure 9: Response Time for range query

global DHT to retrieve the Cluster Master. For this reason, when C is equal to 10, we can notice a lower response delay.

We then analyse the range lookup performance. Specifically, we vary the requested attribute range from [0,10] to [0, 100]. The overall number of connected nodes and the C value (5 attributes per node or 10 attributes per node) are also changed. Results are reported in Figure 9, which shows the response delay versus the size of the range. As expected, when the number of connected nodes is low (light blue and black) the variation in the range does not influence significantly the overall delay. On the other hand, with 100 physical nodes (red and green) the delay increases as the range dimension increases. This can be explained by the fact that more nodes needs to be queried in sequence and in the worst case all nodes in a cluster are queried.

Finally, we mix the range query with the multi attribute feature. To this aim, the number of physical devices is fixed to 100, and the overall number of attributes to 20 and $C = 5$, (620 logical nodes). We perform different requests, varying the number of involved attributes from 5 to 20 (the maximum value) and the range of every attribute. Results are reported in Figure 10. As can be seen, the red line reports the results obtained in the scenario in which 5 attributes are requested simultaneously and, for each attribute, the requested range varies from 0 to X. As expected, the delay significantly

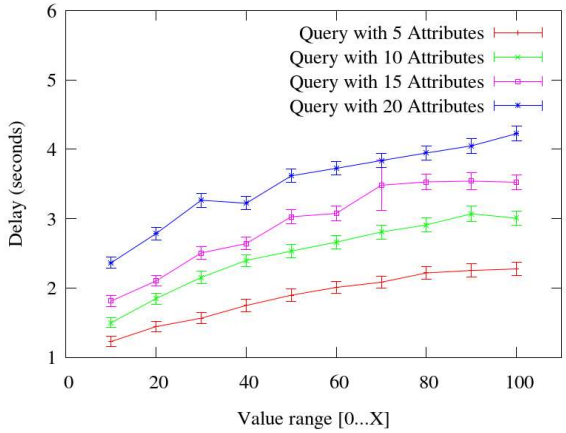

Figure 10: Multi Attribute lookup

increases as the number of involved attributes increases or the requested range increases. However, it is worth to note that, parallelizzation helps in reducing the overall delay for multi-attribute lookup, since doubling the number of attributes does not result in a doubled response delay.

VI. Conclusions

In this work we designed a DHT architecture, with multi-attribute and range queries capabilities, to provide a fog-based distributed lookup service that can be exploited to efficiently discover resources in ITS systems. We design our architecture to dynamically adapt to changing in the environment, both in terms of participating resource attributes as well as in term of participating devices. We assessed the performance of our architecture by means of a prototype and a set of experiments. Results show that the proposed solution scale with the number of nodes allowing to be effectively exploited to provide a distributed lookup service on large scale.

As future work, we plan to extend the proposed solution by exploiting virtualization techniques even inside each device, allowing to migrate logical nodes from one device to another, to enable a load balancing mechanism that can be adapted to dynamic load changes.

References

[1] M. Cai, M. Frank, J. Chen, and P. Szekely, “MAAN: A Multi-Attribute Addressable Network for grid information services”, Journal of Grid Computing, vol. 2, no. 1, pp.3 – 14, 2004.

[2] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of ACM SIGCOMM 2001, 2001.

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In Proc. of ACM SIGCOMM, pages 353–366, 2004.

[4] Shen H, Xu C. Leveraging a compound graph-based dht for multi-attribute range queries with performance analysis. IEEE Transactions on Computers, 2012, 61(4): 433–447

[5] H. Shen and C. Z. Xu, "Leveraging a Compound Graph-Based DHT for Multi-Attribute Range Queries with Performance Analysis," in IEEE Transactions on Computers, vol. 61, no. 4, pp. 433-447, April 2012.

[6] F. Dressler, H. Hartenstein, O. Altintas and O. K. Tonguz, "Inter-vehicle communication: Quo vadis," in IEEE Communications Magazine, vol. 52, no. 6, pp. 170-177, June 2014.

[7] W. He, G. Yan and L. D. Xu, "Developing Vehicular Data Cloud Services in the IoT Environment," in IEEE Transactions on Industrial Informatics, vol. 10, no. 2, pp. 1587-1595, May 2014.

[8] M. Tao, K. Ota and M. Dong, "Foud: Integrating fog and Cloud for 5G-Enabled V2G Networks," in IEEE Network, vol. 31, no. 2, pp. 8-13, March/April 2017.

[9] L.Rizzo, Dummynet: a simple approach to the evaluation of network protocols, ACM SIGCOMM Computer Communication Review, 31-41, 1997

[10] https://linuxcontainers.org/

[11] RFC 3174: US Secure Hash Algorithm 1 (SHA1) (September 2001) by D. Eastlake

[12] Vaquero, Luis M., and Luis Rodero-Merino. "Finding your way in the fog: Towards a comprehensive definition of fog computing." ACM SIGCOMM Computer Communication Review 44.5 (2014): 27-32.

[13] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog computing and its role in the internet of things. In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (MCC '12). ACM, New York, NY, USA, 13-16.