# FOCUS: Scalable Search Over Highly Dynamic Geo-distributed State

Azzam Alsudais[1], Mohammad Hashemi[1], Zhe Huang[2], Bharath Balasubramanian[2],
Shankaranarayanan Puzhavakath Narayanan[2], Eric Keller[1], and Kaustubh Joshi[2]
[1]*University of Colorado, Boulder* [2]*AT&T Labs Research*

***Abstract*—Finding nodes which match certain criteria, based on potentially highly dynamic information, is a critical need in many distributed systems, ranging from cloud management, to network service deployments, to emerging IoT applications. With the increasing scale, dynamicity, and richness of data, existing systems, which typically implement a custom solution based around message queues where nodes push status to a central database, are ill-suited for this purpose. In this paper, we present FOCUS, a general and scalable service which easily integrates into existing and emerging systems to provide this fundamental capability. FOCUS utilizes a gossip-based protocol for nodes to organize into groups based on attributes and current value. With this approach, nodes need not synchronize with a central database, and instead the FOCUS service only needs to query the sub-set of nodes which have the potential to positively match a given query. We show FOCUS's flexibility through an operational example of complex querying for Virtual Network Functions instantiation over cloud sites, and illustrate its ease of integration by replacing the push-based approach in OpenStack's placement service. Our evaluation demonstrates a 5-15x reduction in bandwidth consumption and an ability to scale much better than existing approaches.**

## I. INTRODUCTION

Many distributed systems need the ability to find a node, or a set of nodes, whose attributes match some criteria. A prime example is in cloud management systems, where admins need to identify nodes which satisfy certain properties, such as those that have low CPU utilization, to make scheduling/migration decisions. With the emergence of applications such as edge cloud computing, and Network Service Provider (NSP) deployments [1]–[3] of Virtual Network Functions (VNFs) [4] modern systems are becoming more geographically distributed with more autonomous control within each regional domain. This introduces new challenges around scalability and the need to obtain node attributes directly from the nodes themselves.

Existing approaches, such as those used in cloud management platforms like OpenStack [5], Kubernetes [6], and Mesos [7] can not currently be used beyond the boundaries of a single site because their architectures were not designed for these new requirements. This is because they utilize either a *push* or *pull*-based approach to obtain node information, and neither is sufficient. With *push*-based approaches (used in OpenStack), the nodes periodically *push* their current state through message queues [8] to a central database. Fundamentally, this leads to the centralized database being out of sync with the state as held at the end nodes. Further, as we show in Section III, this approach has limited scalability, requiring applications to work around these limitations (introducing various trade-offs in data freshness, operational complexity and search overhead). In *pull*-based systems, such as used in Google's Borg [9], the controller polls nodes for their current state on demand. This allows for the end nodes to serve as the definitive source of information, but results in expensive communication and ultimately not scalable. While the need for finding nodes that are geo-distributed is recognized as important (e.g., by the OpenStack community [10]), the fact is existing systems simply do not support it.

In this paper, we introduce FOCUS, a scalable service providing timely search across geo-distributed nodes with varied and highly dynamic state. Its design is inspired by scalable peer-to-peer (p2p) systems such as BitTorrent [11]. In particular, central to FOCUS is a gossip-based system where nodes (geographically distributed over the wide area) form groups based on attributes and geographic proximity, which then allows FOCUS to perform directed queries to only the nodes which have the potential to positively respond to the query. We couple this with a query interface which allows FOCUS to be easily integrated into existing applications and support a wide range of complex queries. We demonstrate its flexibility by considering the operational query requirements of a deployed, complex system to instantiate VNFs in an NSP network (Section V-B). We demonstrate its ease of use by replacing equivalent (but not scalable) functionality within OpenStack, for its VM placement service (Section IX).

In summary, this paper describes FOCUS – a novel distributed service for finding nodes, with the following key technical contributions:

- FOCUS provides a query interface which can be easily integrated into larger systems. We demonstrate this by showing how FOCUS can handle complex queries in a production deployment of an NFV service, and replacing OpenStack's message queue based node finding system for placement with a FOCUS-based solution.
- We introduce an approach which enables directed pulls through sortable attribute-based groups. This is scalable and enables end nodes to be the ultimate source of information, as only a subset of nodes might match a query.
- We integrate a gossip-based peer-to-peer coordination into a general distributed application service, which enables end nodes to self-form into groups. This, in turn, alleviates any load on a central component, which in turn, provides much greater scalability.

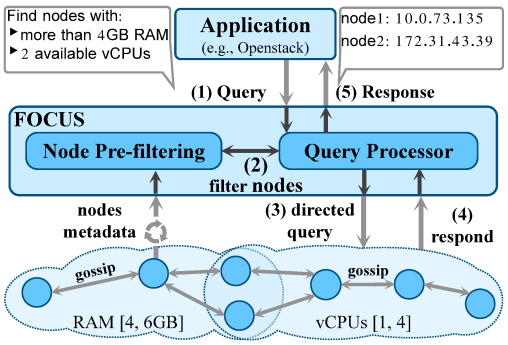



Fig. 1: High-level overview of FOCUS where end nodes form p2p groups based on attribute values (e.g., memory, vCPUs) for scalable query processing. We describe the details of how FOCUS works in later sections (see Section IV for an overview and Sections V-VII for details).

- We implement and evaluate FOCUS in a geo-distributed environment with 1600 simulated nodes, and show a bandwidth consumption reduction between 5x and 15x, when compared to various node-finding techniques.

In the rest of this paper, we first motivate the need for a service like FOCUS through two real world examples (in Section II). We then discuss the limitations of existing solutions in Section III, before describing the architecture of FOCUS, first at a high-level (Section IV), then in detail (Sections V, VI, VII). We then describe our implementation in Section VIII and integration into OpenStack in Section IX. We present our evaluation results in Section X and end with related work (Section XI) and conclusion (Section XIII).

## II. MOTIVATING USE CASES

Searching for nodes with highly dynamic state is a central task of a number of applications. In this section, we highlight two critical applications from our production systems that illustrate the need for a service like FOCUS.

### A. Edge Cloud Management with OpenStack

OpenStack [5] is a cloud management system that helps deploy and maintain virtual infrastructures over large pools of compute, storage, and networking resources throughout a datacenter. To perform this task effectively, in several use-cases in OpenStack there is a need to find physical hosts (among potentially thousands in multi-site and edge deployments [10]) that satisfy certain criteria (exemplified by Table I). For example, the VM Provisioning service will find physical hosts which have enough capacity (RAM, CPU, Disk, Network) to satisfy the needs of the virtual machine that is to be launched. The migration service, which is triggered externally, is an extension of placement and requires similar capabilities to find appropriate hosts. Currently, these services are built around OpenStack agents running on physical hosts pushing status via a message queue [8] to a central database, which limits OpenStack's scalability to barely support thousand hosts [12]. In Section III, we discuss why this is limited in scalability. This limitation is exacerbated in multi-site OpenStack deployments, where multiple hierarchical OpenStack controllers

| Use Cases | Query Examples |
|---|---|
| VM Provisioning / Live Migration | Get hosts that meet new/migrated VM resource requirements |
| Verify Service Status | Get hosts by service type (e.g., compute, scheduler, etc) |
| Tenant Usage Reports | Get hosts belonging to a project ID |
| Hot Spot Detection | Get active/idle hosts |

TABLE I: Example queries from OpenStack source code [13].

are needed to handle the scale and geo-distribution of the deployment [10], introducing more operational complexity.

Clearly, there is a crucial need for a scalable service like FOCUS, that can be queried for physical hosts (*nodes*) matching constraints. In Section IX, we describe how we integrated FOCUS with OpenStack, seamlessly replacing its message-queue-based functionality for placement. Further, with a system like FOCUS, we could go beyond just placement tasks and perform periodic monitoring efficiently (*e*.g., find hosts with a high cache miss rate, indicating that VMs should be migrated). In addition, when using FOCUS within a multi-site deployment of OpenStack, FOCUS can provide a full view of the system (across all sites), reducing the operational complexity of OpenStack imposed by gathering and maintaining node information across all geo-distributed sites.

### B. NFV Automation for Geo-distributed Network Services

The Open Network Automation Platform (ONAP) [1] enables Network Service Providers (NSPs) to automate and support the lifecycle management of complex virtual network functions [4]. Given a network service chain consisting of multiple network functions that traffic must traverse (*e*.g., a firewall and a load-balancer), a key task is to instantiate the chain. This could involve launching new virtual network functions (VNFs), or re-purposing existing VNFs that provide the required service. This task is performed by the ONAP 'homing' service (currently deployed in our production network) that often needs to find sites or service instances that satisfy complex service chain requirements involving a combination of the queries such as the ones in Table II. There could be hundreds and potentially thousands of geo-distributed sites (for edge use-cases) that constitute an NSP's customer premises, central offices and full-fledged datacenters.

To avoid complexity, the homing service currently finds candidate sites and services by executing the above mentioned queries sequentially to various central inventories that only maintain static information (*e*.g., site/service attributes in Table II). Following this step, it hands off the actual task of instantiation to separate processes within the chosen sites (which performs another set of queries, more similar to OpenStack). While the homing service is currently constrained to just static properties, many customers aspire to dynamic properties (*e*.g., site/service capacity).

FOCUS can be a simple replacement for the inventory-sourced querying in the homing service, where each site and service is a 'node' in FOCUS (described in Section V-B). This would enable complex combination of the queries shown in Table II supporting both static and dynamic properties. Alternatively, ONAP's architecture is largely driven by the

| Category | Query Examples |
|---|---|
| Sites | Get all service provider-owned cloud sites |
| Services | Get services of type vGateway, vDNS |
| Site attributes | Sites within 100 miles of a given location and support SRIOV with KVM version 22 |
| Service attributes | vGateways that have the VLAN Tag for the matching customer VPN ID |
| Site capacity | Sites that have a certain tenant quota, available upstream bandwidth, vCPU, available Memory |
| Service capacity | vDNS that can support 10000 resolutions/second, vCDNs that has a hot cache for Customer Y |

TABLE II: Example queries in an operational ONAP deployment.

need to handle a large scale – managing network services across hundreds, soon to be thousands, of sites, where each site has hundreds or thousands of servers. As such, it makes sense to separate functions – the homing service deals with site/service-level constraints and a cloud-level service like OpenStack handles host-level constraints. With FOCUS, we could rethink the architecture wherein the homing service performs both functions, using FOCUS to optimize the search over all hosts/services across sites in a scalable manner.

## III. LIMITATIONS OF EXISTING SYSTEMS

To motivate how FOCUS should be architected, we first examine the way node finding is commonly built into systems today (based around message queues) and why we believe this is not a great match for this purpose. We then discuss some architectural alternatives which could (at a glance) be a solution, but have significant shortcomings as well.

### A. Node Finding with Message Queues

As illustrated in Figure 2a, the approach is broadly characterized by the nodes periodically pushing information about their current status (attributes and current values) to a central database through a message queue. This allows the query processing server to respond to any query to find a node by simply querying the database.

As an example system that is built like this, OpenStack has agents that run on each compute node (usually one per physical host). These nodes each produce a few messages per second containing their current state (*e*.g., number of VM instances, available memory, disk, CPUs, etc) through RabbitMQ [8] (the default messaging queue of OpenStack). A process within OpenStack consumes this information from RabbitMQ and feeds the information into a database.

To quantify the scalability limitation of message queues, we deployed a VM on Amazon EC2 [14], dedicated to run a RabbitMQ server with 8GB of RAM and a CPU with 4 virtual cores (each 2.4GHz), and we used 5 other VMs to host simulated producers (nodes). In each run, we had 100 consumers consuming from 100 queues to which the producers push their messages (we found this to be the most effective way to consume data). Each producer was sending five 1KB messages per second to the server (mimicking OpenStack hosts' behavior). We measured message latency and CPU usage of the RabbitMQ process 30 seconds into the tests.

Figure 3 shows the latency (left y-axis) and CPU usage (right y-axis) when we varied the number of producers from 1K to 8K. As shown, RabbitMQ hits its scalability limit around 6k nodes, and crossed over 50% CPU utilization as early as 2k nodes. While one can argue that adding more RabbitMQ servers can scale the solution, we argue that this not only will consume more resources, but it will also complicate the management of RabbitMQ (multiple RabbitMQ servers need to synchronize through distributed consensus [15], [16]). Such model puts too much emphasis on the messaging queue, making it a bottleneck and a single point of failure. We, therefore, conclude that message queues are not the most efficient means for finding nodes.

### B. Alternate Architectures

Before presenting our gossip-based approach, it is worth considering some alternatives.

*1) Pull:* A first alternative is to pull information from the nodes in response to a query, rather than have the nodes periodically push information. This is illustrated in Figure 2b. When a query comes in, the server can poll the nodes for their current state. The nodes would then send a response to the server, which would process the responses and form a response to the node finding query.

Pull-based approaches are generally not considered scalable, as the server needs to query many nodes simultaneously, and the synchronized responses coming back from the nodes can result in server overload, or problems such as TCP incast [17].

*2) Hierarchy:* It is natural to assume that we could just add hierarchy to address the limitations of push-based message queues or simple pull-based approaches. Here, we consider two approaches to hierarchy and conclude that neither is ideal.

**Aggregating:** Rather than N nodes all sending to a single central server, we can introduce a layer of nodes that simply aggregate the data (as illustrated in Figure 2c). We note that this approach reduces the event rate (number of messages) at the central server, but does not reduce bandwidth consumption, nor does it reduce the event rate at the database.

**Sub-setting:** Rather than push all the way to the central server, we could effectively divide the infrastructure into subsets that each are designed with the nodes all pushing to their subset manager (as illustrated in Figure 2d). Then a central server would query (pull) each of the subset cloud managers anytime a query comes in. This has two key problems. First, this solution partitions the infrastructure, which as has been argued before, is not ideal as crossing the partition boundaries are an added challenge [18]. Second, this inherently increases management complexity – we are now running and managing several cloud managers as opposed to just a single one.

## IV. FOCUS ARCHITECTURAL OVERVIEW

FOCUS, as illustrated in Figure 1, is a system which provides a service to systems that need to find a set of nodes which have certain attributes. Overall, we have two main objectives: (i) serve as a general purpose service for node finding across many applications, and (ii) efficiently scale both in terms of performance and operational complexity. In this section, we highlight the key design/architectural aspects that help achieve this and then elaborate in subsequent sections.

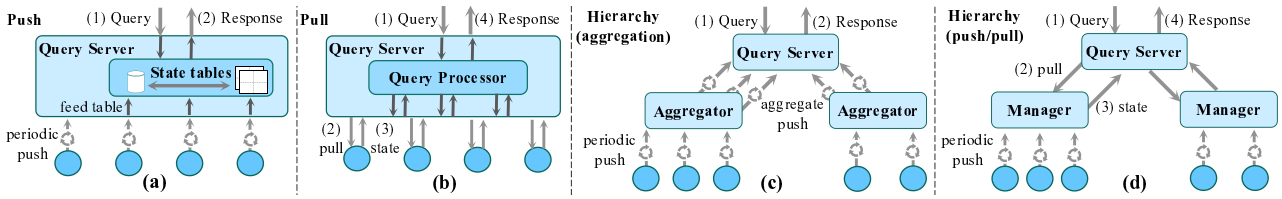



Fig. 2: Various alternate architectural designs that can be used for node finding.

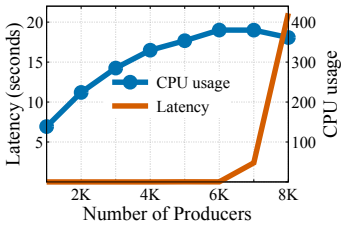



Fig. 3: RabbitMQ test showing latency of messages and CPU usage of the RabbitMQ process while varying the number of producers (i.e., nodes).

**Integrable Query Interface:** In order to be useful to applications, we need FOCUS to have an interface that is easy to integrate and powerful enough to cover a variety of applications' needs. We provide a simple REST API in which a query contains attributes and the range (or specific) values to match. Further, we demonstrate the richness by illustrating the queries needed in a production deployment of NFV services.

**Query Processing with Directed Pulling:** As explained in Section III-B, simple pull-based approaches do not scale beyond a small set of nodes. Yet, at the same time, pulling provides the ability to have the most up-to-date information. To balance between the goals of the scalability (both in terms of performance, and in terms of operational management) and supporting applications with dynamically changing nodes, we introduce directed pulling in FOCUS. Specifically, in our solution, we pre-filter nodes and only send pull requests to nodes which *have the potential* to positively match the query.

**Gossip-based Node Coordination:** To realize directed pulling, we use a gossip-based approach to group nodes based on attribute and value. Crucially, the nodes themselves organize into groups and gossip with each other in order to determine when group membership should change. Then, if needed, they communicate with the central FOCUS node, and change groups. This distributes load from the central server to all nodes, and enables more decentralized decision making. To answer a query, FOCUS simply needs to know which groups a given node is part of (or, said in the inverse, which nodes are part of which groups). Note that, with this approach, we avail all the benefits of the sub-setting approach in Section III-B without incurring any of the operational complexity of maintaining multiple managers.

## V. INTEGRABLE QUERY INTERFACE

A key goal of FOCUS is to be useful across a broad range of distributed applications – *i*.e., it has a query interface that is easily integrated, and rich enough to support the needs of applications. In this section, we first discuss the abstractions, then describe a real-world example.

### A. Abstractions

In this section, we provide a high-level overview of the abstractions provided by FOCUS. As depicted in Figure 1, an application can specify constraints for the nodes it wants to find, and FOCUS will efficiently query the nodes and return nodes (out of possibly thousands) that satisfy the constraints.

**Node Attributes**: Nodes have attributes that can be described as either *static* or *dynamic*. Values of *static* attributes do not change (e.g., number of CPU cores) while values of *dynamic* attributes can and do change over time (e.g., free memory). For multi-site environments, the nodes within a given site inherit the global attributes of that site. For example, a node representing a host in a cloud site will contain not only host attributes such as available CPU and memory, but will also inherit attributes such as "US-East" which describes the cloud site's geographic region.

**Query Structure**: Queries are attribute-oriented, meaning that each application issuing a query should specify the attributes and their desired values. A query structure contains a list of *queryable* attributes, and for each attribute there are the following fields: *name*, *upper bound value*, *lower bound value*, *limit*, and a *freshness* parameter. The attribute *name* is used to describe the attribute of interest to the requester application. The *upper bound* and *lower bound* values are used to support lesser/greater than operations. If an exact match is needed, then both bounds should be of the same value. The *limit* specifies the maximum number of responses to be returned. And finally, the *freshness* field can be specified in terms of milliseconds (a value of zero means the response must be as close to real time as possible to guarantee extremely fresh results). We note that this is one version of a query structure, and there are multiple versions that FOCUS supports for other attribute types (e.g., location, text-based attributes, etc).

### B. Example Queries used in VNF Homing

To illustrate the use of FOCUS, here we consider an operational example of the VNF homing service described in Section II-B. In this example, we specifically present the case which matches today's use (first searching for sites and services, and then performing instantiation), rather than re-architecting the solution which could be enabled by FOCUS (searching for physical hosts and services across sites and deploying a service chain in a one step process).

Figure 4 shows the homing requirements of a virtual Customer Premises Equipment (vCPE) [19] network service, that provides residential broadband connectivity. Figure 4a shows the layout architecture, connecting the residence to the vG (virtual gateway) hosting infrastructure at the Service Provider Edge (PE). Here, the bridged residential gateway (BRG) is the

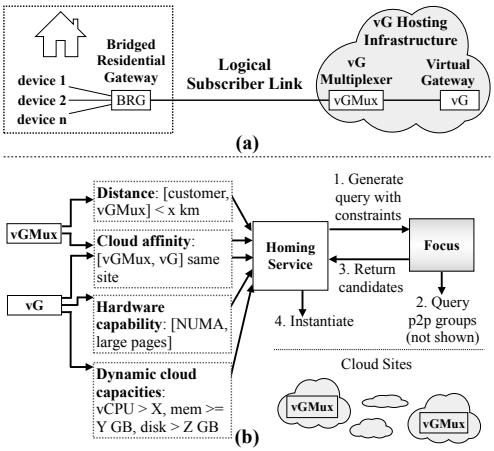



Fig. 4: VNF homing: an apt use-case that illustrates the use of FOCUS for homing the residential virtual Customer Premises Equipment (vCPE) service [19] in ONAP [1].

vCPE located at the residential customer premises, while the vG Multiplexer (vGMux) is a shared network function at the PE that maps layer-2 traffic between a subscriber's BRG and its unique vG, ensuring traffic isolation between customers.

Homing the vCPE service requires finding a slice of an existing vGMux instance and finding a suitable cloud site for spinning up a new instance of the vG. Figure 4b shows the homing policies (or constraints) that drive the selection of an optimal vGMux and the corresponding PE site to host the vG for a given customer. While the first two constraints are relatively static, the hardware capabilities of a cloud site may change as new host aggregates are added, and instantaneous site capacities may vary at even shorter time scales since resources are typically shared among multiple services and customers. As shown in Figure 4b, FOCUS is a perfect fit for this problem wherein those constraints can be expressed as a query to FOCUS which will return a set of candidate vGMux instances and sites (FOCUS 'nodes') that satisfy all constraints.

## VI. QUERY PROCESSING WITH DIRECTED PULLING

In this section, we discuss the key concept of grouping nodes based on their attribute values, how FOCUS is able to pull from the right subset of nodes, and key optimizations for FOCUS's query processor.

**Attribute-based Grouping**: As demonstrated in Section V, the state of each node is described in the form of attributes (e.g., CPU utilization, free memory and disk, location, etc). Naturally, grouping nodes based on their attribute values makes it possible to filter out many nodes that cannot satisfy certain queries. Driven by our query structure, we group nodes based on attribute values that are within a specific range. For example, in Figure 1, there are two groups of nodes – one for nodes with free RAM of 4 to 6GB, and another for nodes with 1-4 virtual CPUs. Note that a node can be in multiple groups simultaneously (e.g., having 5GB of RAM and 2 vCPUs). In Section VII, we discuss how nodes form such groups in a dynamic manner, adapting to changing attribute values.

**Query Conjunctions through Sorted Pulls**: Having pre-filtered nodes and prior (coarse-grained) knowledge of the current state of each node in the system, it is possible for FOCUS to direct queries to only those nodes that have the potential to satisfy the queries. Specifically, when FOCUS receives a query, it parses the query and sends it to the corresponding groups that satisfy the query conditions. The members of the group, then, will respond with their current state. For instance, consider a query to retrieve nodes with 4GB of RAM. FOCUS will send the query *only* to the group that has 4 to 6GB of free RAM (exemplified in Figure 1).

Multi-attribute/constraint queries, if not handled well, can undermine the advantages of pre-filtering and attribute-based grouping. That is, if a query containing too many constraints for different attributes is sent to every single group of nodes that correspond to each attribute, then this can quickly degenerate to the case where the query is sent to every single node in the system. Instead, FOCUS sends the query to the smallest group that corresponds to one of the query's attributes (mechanism described in Section VII). Then, the nodes within that group can answer to all constraints in the query. This narrows down the scope of nodes to which a query must be sent even further.

**Optimizations**: In addition to sending queries to the smallest group, we further optimize our querying with a cache to store query responses along with a timestamp of when they were fetched. Checking the cache is the first step in processing a query. As described in Section V, each query has a freshness parameter, which is checked against responses fetched from the cache. Should cached responses not qualify for the query freshness or in the event of a cache miss, the query will be sent to the appropriate group. Moreover, under heavy-load conditions, and after determining what groups to which the query must be sent, FOCUS has the mechanism to delegate to the querying application the act of actually sending the query to the nodes. As a result, the load on FOCUS is alleviated, making it more lightweight and scalable. However, responses for those delegated queries will not traverse FOCUS, and consequently will not be cached.

## VII. GOSSIP-BASED NODE COORDINATION

Our attribute-based groups are p2p groups that implement the gossip protocol [20], through which each node gossips membership information with only a few members of the group, who in turn gossip with other nodes until convergence in reached. The gossip channel is also used to disseminate queries received from the FOCUS server. In this section, we describe how nodes form those groups and how FOCUS is able to maintain information that is later used to process queries.

**Dynamic Groups Management**: Upon registering with FOCUS, a node reports its current attributes and the corresponding values. In return, FOCUS will provide entry points into the appropriate groups for the node to join. Each of the node's dynamic attributes corresponds to a specific p2p group based on the attribute value[1]. When there are no existing groups

[1]Static attributes are maintained in the FOCUS distributed data-store (described in Section VIII) and therefore do not need to be managed via groups.

that suit the new values of a node, FOCUS will instruct that node to start a new group and, in turn, will be an entry point for future nodes. In addition to providing entry points, group suggestions from FOCUS also contain group ranges. A group range is used by nodes to detect when it is necessary to move to other groups (when new values fall outside the group range). Based on predefined attribute value cutoffs, FOCUS decides the range of each attribute-based group.

Further, in order to keep groups from growing indefinitely, which as we show (in Section X) has an impact on query latency, FOCUS keeps track of how many members are in a group. When a group size exceeds a certain threshold, FOCUS will fork groups by suggesting new groups to new nodes. We note that a group that tracks an attribute spanning multiple geographic locations (e.g., free RAM) could be formed disregarding the geographic locations of the nodes in the group, which could degrade performance. However, we can seamlessly split groups when they exceed certain geographic thresholds (like maximum distance among nodes) by treating them as separate attributes tied to location. For example, if a group containing nodes that have more than 4GB of free RAM traverses locations across say Texas and California, we simply create two groups: (nodes with more than 4GB of free RAM in Texas) and (nodes with more than 4GB of free RAM in California). Accordingly, when FOCUS processes a query that does not specify a location (e.g., *get nodes with greater than 4GB of RAM*), FOCUS will query groups with nodes having more than 4GB of RAM in all locations (or until it satisfies the *limit* parameter in the query as described in Section V-A), aggregate the results, and return them to the query initiator.

**Group Member List through Representatives**: FOCUS maintains a list of member nodes for each group to serve as entry points for other nodes, to enable queries, and to perform operations like forking the group based on size, geography, etc. Even if this list is relatively stale, as long as this list includes some reachable, live member, FOCUS can pull the latest attribute values for the group (explained later in this section). To obtain this list, FOCUS randomly selects a small (configurable) number of nodes in each group to be the representatives and asks them to periodically upload the group member list. Since modern gossip protocols exchange and construct member lists, representatives nodes need to do minimal additional work in uploading this information. Further, the randomized representative node selection ensures that the workload is distributed evenly. In a group that has high churn rate, more representative nodes and/or more frequent updates are required. Finding the optimal upload frequency for different systems is an important aspect of future work.

**Load-balanced Query Routing within p2p Groups**: To receive a query response from a p2p group, FOCUS can send the query to any member of the group which, in turn, will gossip the query with other members of the group. Query responses from group members, however, will be directly sent to the member who originated the query in order to allow fast query processing. A key design decision of FOCUS is that the load must be distributed across all nodes. Further,

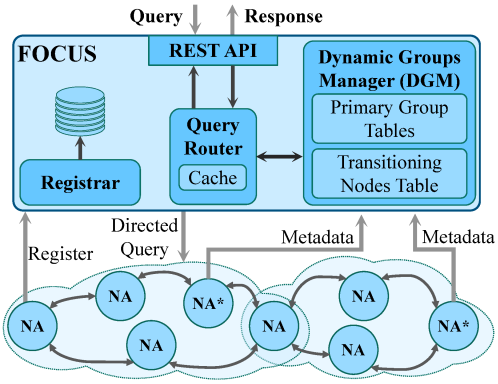



Fig. 5: Internal design of FOCUS, showing node agents (NA) while group representatives (denoted with *) push their groups' metadata to FOCUS.

FOCUS needs to be resilient to failure of nodes in the attribute groups. Therefore, every time a query needs to be sent to one of the p2p groups, FOCUS randomly picks a different group member, as opposed to just sending them to the representatives described above. In section XII, we describe future work to address the efficiency/communication trade-offs in this design.

Since node coordination is gossip-based, convergence can be relatively slow and if a query is sent to a group immediately after a new node has joined, the new node may not receive the query. To solve this problem, FOCUS maintains a table in its data-store that tracks nodes that are transitioning between groups or are entering a new group (when they ask FOCUS for group suggestions). To ensure inclusiveness, FOCUS also includes nodes in this table when processing a query.

## VIII. IMPLEMENTATION

In this section, we describe the internal design of FOCUS (Figure 5) and provide details of its implementation. Specifically, we implemented FOCUS in Java with 3.1K lines of code (1.9K LoC for the FOCUS service and 1.2K LoC for the node agent). We used *Apache Cassandra* [21] as our service data store, *Eclipse Jetty* [22] as our web server, and *HashiCorp's Serf* [23] as our p2p fabric. First, we describe each component of the FOCUS service and then describe the node agent.

### A. FOCUS *Service*

Each of the three main components of FOCUS (Registrar, Dynamic Groups Manager (DGM), and Query Router) exposes its services through a REST API that is hosted on a *Jetty* server. The input and output of each API call is JSON-formatted [24].

*1) The Registrar:* The Registrar listens for node registration requests. Each request contains certain information about the new node, including: node IP address, a list of attribute-value pairs, and the port through which the node can receive commands and queries from FOCUS. The Registrar stores the new node information at the FOCUS database, which is backed by a Cassandra cluster to provide resiliency and fault tolerance. For each *static* attribute, the Registrar creates a table containing: node ID, attribute value, and a timestamp field. We also use an additional field to store all other attribute-value pairs for each node. For instance, if the new node has the

following *static* attributes (*arch*:*x86*, *cores*:*8*), then an entry at the table for the *arch* attribute will look like the following.

| node ID | arch | attributes | timestamp |
|---|---|---|---|
| IP address | x86 | {cores:8} | time value |

Storing other attributes in each attribute table makes it much more efficient to query the database. That is, to perform a query with multiple attributes, we just need to query one table, the table with the lowest number of entries. These tables are also updated by the DGM when it learns new information.

*2) The Dynamic Groups Manager (DGM):*

**Deterministic Group Naming**: Choosing consistent group IDs is crucial when referring new nodes to groups as well as when routing queries to the desired group. The DGM, using a deterministic group naming function, constructs group names using an attribute cutoff. For instance, if the *disk* attribute cutoff is set to 10, then a group named *disk.10GB* will contain nodes that have between 10 and 20 GB of free disk space. Our deterministic group naming function accepts an attribute-value pair, and returns the corresponding group name.

**Group Tables**: The DGM keeps track of groups by storing them in a *primary* key-value lookup table, which frequently gets synchronized with the Cassandra data-store. We note that since group information is essentially maintained by the groups themselves, failure recovery of the DGM comes naturally. That is, when the DGM fails and a new one is instantiated, group representatives will send their corresponding group information, which the new DGM uses to populate its *primary* group tables. As discussed in Section VII, information about nodes transitioning between groups will be kept in a temporary table until they appear in one of the groups updates.

*3) The Query Router:* In order to separate the load between the northbound API (consumed by querying applications) and the southbound API (consumed by nodes), we bind the Query Router to a different port than the DGM. It runs a process that has a cache table in memory, which is checked every time a query is received. For queries with only *static* attributes, the Query Router will get the corresponding values directly from the database. Otherwise, it will send the query to the corresponding group after consulting the DGM. To prevent FOCUS from indefinitely blocking on queries, FOCUS uses a configured timeout after which the query processing will abort.

### B. Node Agents

Our node agent consists of two light-weight processes: a node manager and a p2p agent, both of which run on every node in our system. The node manager is responsible for communicating with the FOCUS service for managing node registration and requesting group suggestions. And the p2p agent (which runs a Serf client [23]) is responsible for connecting to other p2p agents for each of the node's attribute groups, one group per attribute.

**Node Manager**: The node manager has three tasks. (*i*) It runs OS commands that collect resource (attribute) information (CPU usage, free RAM and disk, etc). (*ii*) It handles communication with the FOCUS service. (*iii*) It provides an interface for receiving queries and commands (e.g., representative election) from the FOCUS service. When a node receives a query, the node manager will gossip the query (via its p2p agent) to the members of its group and gets the response back.

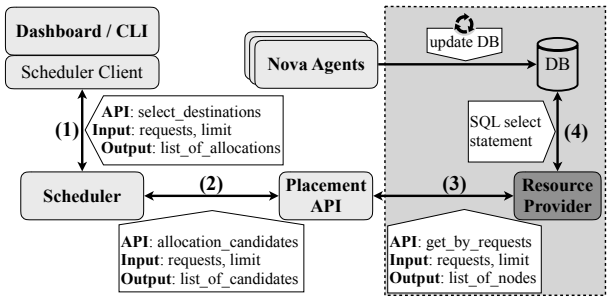



Fig. 6: Order of API calls within OpenStack for provisioning a VM instance. The shaded area shows where FOCUS is integrated by replacing the object that queries the database with a FOCUS client that queries the FOCUS service.

**p2p Agents**: Each node runs a separate Serf agent for each group it joins. When a node requests group entry information from FOCUS, it will get a list of entry points for each group. Each entry point consists of the IP address and the port number at which the Serf agent of the node is listening. The requesting node can use this information to join the p2p group. In the case of first node to register for a group, FOCUS will let the node know there are no entry points; consequently, the node will start a Serf agent and immediately let FOCUS know about its Serf binding port so that future nodes can join. Note that, the Serf agent is configurable with a set of parameters, including: the number of neighbor nodes to gossip with (gossip fanout) and a gossip interval parameter. In our implementation of the FOCUS node agent, we set the fanout to 4 nodes and the gossip interval to 100 milliseconds[2]. This setting achieves a balance between overhead on the node agents and query performance.

## IX. OPENSTACK INTEGRATION

In this section, to demonstrate FOCUS's usability, we provide a detailed overview of how we integrated FOCUS into OpenStack's VM placement service (using OpenStack version 3.15.0 and Nova version 18.0.0), thereby providing a much more scalable solution compared to RabbitMQ (as described in Section III). First, we provide an overview of how OpenStack finds nodes for new VM placements (and live migration), which is illustrated in Figure 6.

**Finding Nodes for VM Placement**: OpenStack follows a model that resembles the one in Figure 2a, where Nova compute nodes running on each physical host periodically push their state updates to a centralized database through RabbitMQ, containing information about current capacities (cpu, ram, disk, etc) and virtualization-specific information (number of installed VMs, number of vCPUs, etc). Each VM placement request object takes the following form.

```
struct{ int limit, dict resources}
```

The `limit` field is used to limit the number of nodes in the response. The dictionary of `resources` contains the minimum required resources for the requested VM image.

[2]This allows a 400-node group to reach convergence in as little as 0.6 sec.

Such resources typically are: `RAM` (specified in Megabytes), `Disk` (specified in Gigabytes), and `VCPUs` (an integer value specifying the number of required virtual CPUs).

When a VM placement request is issued, the following steps take place (in accordance with steps in Figure 6). **(1)** A scheduler client (mainly used by the dashboard or CLI) will call the scheduler API `select_destinations` by passing the requested resources and a limit for the desired number of placement candidates. **(2)** The scheduler, in turn, will verify the request and then call the Placement API GET method `allocation_candidates`, which returns a list of placement candidates so that the scheduler can issue commands to the desired candidates to spawn a new VM. Upon receiving a placement request, the Placement service **(3)** calls the Resource Provider's `get_by_requests` method, which **(4)** queries the database and returns a list of candidates.

**Integrating FOCUS**: The `allocation_candidate` class of the placement service makes an indirect call to the database in order to fetch the available hosts and their state. The following line of code makes the request.

```
cands = rp_obj.AllocationCandidates
      .get_by_requests(requests,limit)
```

We replace this particular functionality, corresponding to the shaded box in Figure 6 for steps (3) and (4) with a FOCUS-based solution. Specifically, we replace the above call to the central database with the following single call to FOCUS.

```
cands = fc_obj.query(requests,limit)
```

Where `fc_obj` is an instance of a class that we implemented to handle queries from OpenStack to FOCUS. Currently, it supports placement queries, and adding support for other queries merely requires adding more functions to this class.

**Augmenting FOCUS's Node Agents**: We augmented our node agents (that now run on the physical hosts running the Nova compute agents) with the `libvirt` virtualization library [25] in order to gather resource information. Our node agent interfaces with `libvirt` and connects to the QEMU hypervisor [26] to gather the required information. This addition to our node agent resulted in less than 100 lines of additional code to the original node agent code. Although our current integration connects to the QEMU hypervisor, we can easily integrate with other hypervisors (Xen [27], KVM [28], VMWare ESX [29], etc) that `libvirt` supports.

## X. EVALUATION

In this section, we evaluate the performance of FOCUS and compare it against different node finding approaches. Our evaluation of FOCUS answers the following questions:

1) How does FOCUS scale compared to other solutions?
2) How efficiently does FOCUS perform with real-world query traces?
3) What are the FOCUS benchmarks with respect to group size, and overhead on the node agents?

### A. Testbed Setup

To evaluate the performance of FOCUS, we deployed it on Amazon's EC2 [14] and to simulate geo-distributed nodes, we chose four different EC2 regions in North America: Ohio, Canada, Oregon, and California. In each region, we instantiated 8 VMs each with 16GB of memory and 4 vCPUs to host our FOCUS node agents. Since multiple node agent programs are consolidated onto the same VM in our experiments, we introduced a randomness factor[3] to the node agents which they use to change their attribute values so that they do not report the same information of the VM on which they run. Each node agent reported 4 attributes: *CPU usage*, number of available *vCPUs*, free *RAM_MB*, and free *DISK_GB* space. The attribute-based group cutoffs were as follows: {CPU usage: 25%, vCPUs: 2, RAM_MB: 2048MB, disk: 5GB}. This means that nodes with CPU utilization between zero and 25% will be in the same group, and nodes that have 1 to 2 virtual CPUs will be in the same group, and so on.

### B. FOCUS *vs. Existing Systems*

**Bandwidth Consumption**: In this experiment, we evaluate FOCUS's scalability by measuring the bandwidth consumption at the query server. We also compare our results with the following node finding approaches. (*i*) Naive push and pull, where node state is either frequently pushed from the nodes (Figure 2a) or pulled on-demand (Figure 2b). (*ii*) Static hierarchy (Figure 2d), where the number of state managers is 16.[4] We also compare against (*iii*) RabbitMQ with two configurations (publish and subscribe), where nodes either periodically publish information (pub) or subscribe for queries (sub) and then respond. The query/update frequency is 1/second.

Figure 7a shows that FOCUS consumes less bandwidth than other systems. For instance, when the number of nodes reaches 1600, FOCUS can eliminate up to 86%, 92%, 93%, and 95% of the communication between the server and nodes when compared to static hierarchy, RabbitMQ (pub), naive push/pull[5], and RabbitMQ (sub), respectively. This shows that FOCUS, with attribute-based grouping and directed pulling, can scale much better than other approaches.

**Query Processing Latency**: Figure 7b shows the average query latency for FOCUS when compared to RabbitMQ while processing 40 queries per second. Note that FOCUS was deployed according to the setup described earlier in Section X-A; however, our RabbitMQ deployment was in one region of EC2 where we ran a RabbitMQ server and multiple simulated producers. Up to 1K nodes, RabbitMQ shows faster responses. However, after 1K nodes, RabbitMQ could not scale, while FOCUS's latency stays relatively constant. This is because instead of sending queries to all nodes, FOCUS used directed pulling to send queries only to the corresponding p2p groups.

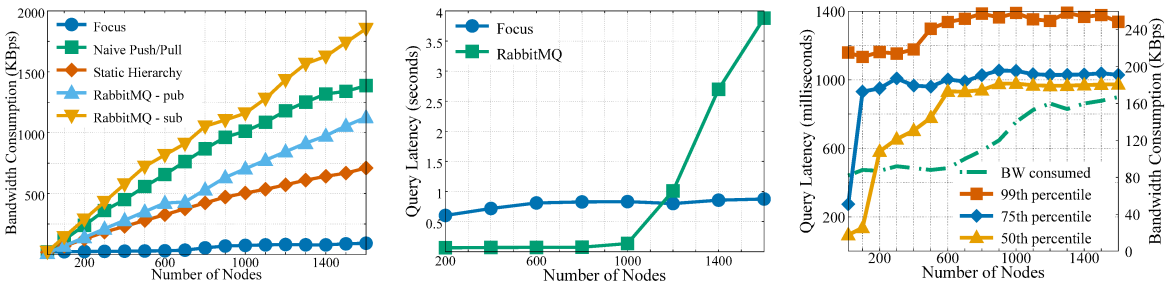
### C. Query Latency for Real-world Traces

In order to get a sense of how well FOCUS can perform in real-world deployments, we replayed a cloud trace from the Chameleon cloud testbed [30], containing OpenStack KVM

[3]The randomness factor depends on the attribute value range. E.g., the value for *cpu usage* can be randomly assigned from 0 to 100.

[4]We chose 16 because that was the average number of group representatives that are in charge of reporting group information to FOCUS.

[5]Naive push and pull showed identical results; hence, merged into one line.




(a) Scalability in terms of bandwidth consumption of the query server for multiple node finding approaches.

(b) Average query latency (*s*) for FOCUS and RabbitMQ, when processing 40 queries per second.

(c) Query latency (*ms*) for real-world queries with nodes deployed across 4 geographical regions.

Fig. 7: FOCUS's performance for different metrics when compared to other systems and when processing real-world queries.

events. The trace contains over 75K VM placement events over the course of 10 months. Those events provide resource requirements, which we parsed into our queryable attribute object (described in Section V), and then replayed those queries to FOCUS. We replayed the traces at an accelerated rate (15,000x faster) to test how well FOCUS performs under heavy loads. All queries were sent to the p2p groups, and the cache was disabled for this experiment.

Figure 7c shows the latency per request for the 50th, 75th, and 99th percentile of queries (left y-axis). The results show that there is a steady increase in the response latency until the number of nodes reaches about 600 nodes, after which the latency stays relatively constant. We monitored the FOCUS service at each run, and found that after 600 nodes, the average group size did not significantly increase (≈150 members per group), whereas the number of groups kept increasing. This highlights the benefit of using an attribute-based grouping.

### *D. Microbenchmarks*

In this section, we provide microbenchmarks of various aspects that impact FOCUS's performance.

**Resource Usage of the FOCUS Server** While replaying real cloud traces (discussed in Section X-C), we measured the CPU and RAM usage of the FOCUS server. Figure 8a shows the resources used by the FOCUS server while we increase the number of nodes. The results highlight that the FOCUS server is not resource-hungry, even when there are more than 1.5K nodes that are part of the FOCUS system. Note that the VM on which the server ran had 4 virtual CPUs and 16GB RAM.

**Overhead on Node Agents**: Figure 8b shows the bandwidth consumption at one of the nodes in a p2p group under two conditions: normal operation (exchanging membership information) and query processing (1 query/s). Even though that in our earlier experiments the average size of a group did not exceed 150 members, it is worth measuring the impact of having groups with hundreds of members. The results in the figure suggest that during normal operation, the bandwidth consumed is negligible (under 2KBps), even for groups with more than 400 members. This shows that even when deployed in a cross-site setting, FOCUS does not impose significant overhead on the nodes. When processing queries every second, the bandwidth consumed by the node is less than 10KBps for groups with 100 nodes and about 50KBps for groups with 400 nodes. We note that FOCUS picks a random member every time it sends a query to a group; hence, eliminating any excessive overhead on one specific node.

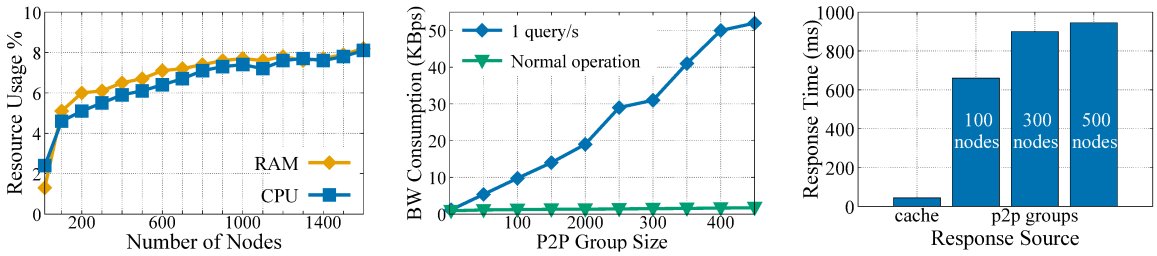
**Latency vs. p2p Group Size**: Since queries are gossiped by the p2p nodes, query processing times depend mainly on how fast a p2p group can converge. In this experiment, we measure the query latency with respect to the source of the query response from which it was served. In our design of FOCUS, query responses are either fetched from a local cache or pulled from the p2p groups. For responses from the p2p groups, we measured the start and end times at the p2p node that received the query. Figure 8c shows two key points. First, fetching responses from the cache (takes 45ms) significantly reduces the query processing time by an order of magnitude. Second, even when a p2p group contains hundreds of nodes, the response is still under one second. Note that we used the same gossip configuration discussed in Section VIII-B.

## XI. RELATED WORK

FOCUS is an extension of our earlier work [31] which introduced the idea to use p2p groups. In this paper, we provide the complete architecture, implementation, evaluation and details of OpenStack integration.

**Centralized Lookup Services.** Node finding is a fundamental problem in any networking system. For example, name-based networking solutions such as the Intentional Naming System (INS) [32], Auspice [33]–[35], the Global Naming Service [36] or the geographic-addressing [37] in FocusStack [38] all propose to implement a centrally managed resolution service with nodes pushing updates to the centralized lookup service. Across many domains, either a push or a pull-based approach is used to enable the central service to satisfy the lookup – with the trade-offs having been carefully studied in [39]. In contrast, FOCUS follows a hybrid approach wherein a list of group members is periodically pushed to the centralized lookup service while to find the list of nodes satisfying a query, FOCUS uses a directed pull-based approach.

**p2p Lookup Services.** Node lookups are a fundamental part of p2p services. Systems like Gnutella [40] used a flooding protocol for information dissemination, Kademlia [41] uses a structured distributed hash table that allows node look up through structured IDs, and the gossip algorithm in [42]




(a) CPU and RAM usage of the FOCUS server while processing queries at 40 queries per second.

(b) Node bandwidth (BW) consumption under two conditions: normal operation and query processing every second.

(c) Query response latency for different response sources: cache or p2p groups of different sizes.

Fig. 8: Various microbenchmarks showing different aspects of FOCUS's performance.

builds an unstructured p2p network and balances the update workload evenly among all the members in the group. The key differentiator in FOCUS is that a centrally managed system is dynamically managing the value-based p2p groups (based on Serf [23]) aiding nodes in joining and leaving the groups, and allowing for smaller p2p groups (reducing convergence time, and by extension, faster responses to queries).

**Attribute-based Grouping.** A key design decision in FOCUS is to group nodes in terms of attributes. This is similar in approach to publish-subscribe (pub-sub) systems [8], [43]–[46], which also provide attribute-based grouping (e.g., channels and topics). But such systems, too, will not scale because nodes need to constantly publish or notify subscribers of their state through a global queue server (a bottleneck), whereas in FOCUS we use a directed pull approach.

**Cloud and Cluster Management.** FOCUS's scalable and loosely-coupled design provides cloud and cluster management platforms (e.g., OpenStack [5], Google's Borg [9], Kubernetes [6], etc) with scalable search and a comprehensive view of the system with close to real-time information as compared to their push-based (OpenStack and Kubernetes) or pull-based approaches (Borg). Further FOCUS minimizes the resource usage of the controller. For instance, in order to scale Kubernetes to more than 500 nodes, the controller needs to have at least 36 vCPU cores and 60GB of RAM [47]. FOCUS's server, on the other hand, needs only 4 vCPU cores (an order-of-magnitude lower) and 16GB of RAM, out of which FOCUS utilizes only 10% to manage 1600 nodes (Figure 8a).

## XII. Discussion and Future Work

In this section, we discuss some of the open issues in FOCUS and potential future directions of work.

**Deciding the Right Group Ranges**: Determining the "right" group ranges (Section VII) is critical for FOCUS's performance as biased groups could form and harm FOCUS's ability to efficiently answer queries. Our design allows system operators to configure group ranges, allowing them to use the method of their choice (static, random, heuristic, trace-driven, etc). In the future, we will augment FOCUS with a default mechanism driven by machine learning techniques (trained with traces) to decide appropriate group ranges.

**Faster Query Processing**: In our evaluation (Figure 7b), even though FOCUS scales better than RabbitMQ, FOCUS's query processing takes longer than what RabbitMQ takes for nodes less than 1200. This is mainly attributed to FOCUS's use of a small number of nodes as its "fanout" gossip factor (e.g., number of neighbors that a node directly talks to), leading to a slow convergence. Another alternative (and faster) approach is to broadcast the query to all members of the group by configuring the fanout factor to be N, where N is the number of nodes in the group. This is a trade-off between resource usage of a node (bandwidth consumption used to gossip with other members) and query processing latency. In the future, FOCUS can provide the option to configure each group's fanout factor, which when set to a high value, will be of great use for time-sensitive applications.

Further, as future work, we first wish to explore materialized views in FOCUS by creating specific p2p groups representing frequently issued queries. We wish to extend this concept by supporting event triggers – change in node state will automatically update the materialized view. Finally, we also wish to provide translation/normalization functions to deal with the potential heterogeneity in data sources of FOCUS.

## XIII. Conclusion

In this paper, we address a fundamental problem in large-scale distributed systems – finding nodes that match certain criteria and present FOCUS, a novel scalable search service for finding nodes. This is a challenging problem because of the scale of the nodes and the highly dynamic nature of their attributes. Current approaches to this problem that typically involve nodes pushing status to a centralized database through a message queue simply do not scale. Naiive hierarchical push/pull solutions impose unsuitable trade-offs in accuracy of the results and overhead of node finding. On the other hand, FOCUS uses a novel hybrid approach in which we maintain p2p groups of nodes based on their attribute values or state. We illustrate FOCUS's broad applicability in real-world systems such as OpenStack and VNF homing in the Open Network Automation Platform (ONAP). Our evaluation confirms the superior scalability of FOCUS over existing approaches.

## Acknowledgements

This research was supported in part by the National Science Foundation under grant 1652698 (CAREER). We would also like to thank the anonymous reviewers for their insightful feedback.

## REFERENCES


[1] ONAP, "Open Network Automation Platform," https://www.onap.org/wp-content/uploads/sites/20/2017/12/ONAP_CaseSolution_Architecture_120817_FNL.pdf.

[2] "Central Office Rearchitected as a Datacenter (CORD)," http://opencord.org/wp-content/uploads/2016/03/CORD-Whitepaper.pdf, Mar. 2016.

[3] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[4] "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action," http://portal.etsi.org/NFV/NFV_White_Paper.pdf, 2012.

[5] "Openstack: Open source software for creating private and public clouds," https://www.openstack.org.

[6] "Production-Grade Container Orchestration - Kubernetes," https://kubernetes.io.

[7] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972488

[8] "Rabbitmq," https://www.rabbitmq.com/.

[9] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[10] "Openstack cascading solution," https://wiki.openstack.org/wiki/OpenStack_cascading_solution.

[11] "BitTorrent," http://www.bittorrent.com.

[12] "OpenStack Scalability Tests," https://docs.openstack.org/developer/performance-docs/test_results/1000_nodes/\index.html.

[13] "OpenStack Nova," https://github.com/openstack/nova.

[14] "Amazon Elastic Compute Cloud (EC2)," https://aws.amazon.com/ec2/.

[15] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference (ATC)*, 2014, pp. 305–319.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX Annual Technical Conference (ATC)*, vol. 8, no. 9. Boston, MA, USA, 2010.

[17] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP Incast Throughput Collapse in Datacenter Networks," in *Proc. ACM Workshop on Research on Enterprise Networking (WREN)*, 2009.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM computer communication review*, vol. 39, no. 4. ACM, 2009, pp. 51–62.

[19] "Network Enhanced Residential Gateway," https://www.broadband-forum.org/technical/download/TR-317.pdf.

[20] A. Das, I. Gupta, and A. Motivala, "Swim: Scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 303–312.

[21] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[22] "Eclipse Jetty," https://www.eclipse.org/jetty/.

[23] "Serf: Decentralized Cluster Membership, Failure Detection, and Orchestration," https://www.serf.io/.

[24] "JSON (JavaScript Object Notation)," http://www.json.org/.

[25] "libvirt: The Virtualization API," https://libvirt.org/.

[26] "QEMU, the FAST! processor emulator," https://www.qemu.org/.

[27] "The Xen Project, the powerful open source industry standard for virtualization," https://www.xenproject.org/.

[28] "Kernel Virtual Machine (KVM)," https://www.linux-kvm.org/page/Main_Page.

[29] "ESXi — Bare Metal Hypervisor — VMware," https://www.vmware.com/products/esxi-and-esx.html.

[30] "Chameleon Cloud: A configurable experimental environment for large-scale cloud research," https://www.chameleoncloud.org/.

[31] A. Alsudais, Z. Huang, B. Balasubramanian, S. P. Narayanan, E. Keller, and K. Joshi, "Nodefinder: scalable search over highly dynamic geo-distributed state," in *10th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[32] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The design and implementation of an intentional naming system," *ACM SIGOPS Operating Systems Review*, vol. 33, no. 5, pp. 186–201, 1999.

[33] A. Sharma, X. Tie, H. Uppal, A. Venkataramani, D. Westbrook, and A. Yadav, "A global name service for a highly mobile internetwork," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 247–258, 2015.

[34] A. Venkataramani, J. F. Kurose, D. Raychaudhuri, K. Nagaraja, M. Mao, and S. Banerjee, "MobilityFirst: A Mobility-centric and Trustworthy Internet Architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 74–80, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656888

[35] A. Venkataramani, A. Sharma, X. Tie, H. Uppal, D. Westbrook, J. Kurose, and D. Raychaudhuri, "Design requirements of a global name service for a mobility-centric, trustworthy internetwork," in *2013 Fifth International Conference on Communication Systems and Networks (COMSNETS)*, Jan 2013, pp. 1–9.

[36] B. W. Lampson, "Designing a global name service," in *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*. ACM, 1986, pp. 1–10.

[37] R. J. Hall, "A geocast-based algorithm for a field common operating picture," in *MILITARY COMMUNICATIONS CONFERENCE, 2012-MILCOM 2012*. IEEE, 2012, pp. 1–6.

[38] B. Amento, B. Balasubramanian, R. J. Hall, K. R. Joshi, G. Jung, and K. H. Purdy, "FocusStack: Orchestrating Edge Clouds Using Location-Based Focus of Attention," in *IEEE/ACM Symposium on Edge Computing, SEC 2016, Washington, DC, USA, October 27-28, 2016*, 2016, pp. 179–191. [Online]. Available: https://doi.org/10.1109/SEC.2016.22

[39] M. Bhide, P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, "Adaptive push-pull: disseminating dynamic Web data," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 652–668, Jun 2002.

[40] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker, "Making gnutella-like P2P systems scalable," in *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 407–418. [Online]. Available: http://portal.acm.org/citation.cfm?id=863955.864000

[41] P. Maymounkov and D. Mazieres, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[42] A. Das, I. Gupta, and A. Motivala, "SWIM: scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*, 2002, pp. 303–312.

[43] G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a Replicated Logging System with Apache Kafka," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1654–1655, Aug. 2015. [Online]. Available: http://dx.doi.org/10.14778/2824032.2824063

[44] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman, "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, ser. ICDCS '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 262–. [Online]. Available: http://dl.acm.org/citation.cfm?id=876891.880590

[45] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[46] "Distributed Messaging - zeromq," http://zeromq.org/.

[47] "Building Large Clusters - Kubernetes," https://kubernetes.io/docs/setup/cluster-large/.