NodeFinder: Scalable Search over Highly Dynamic Geo-distributed State

Azzam Alsudais[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 with certain criteria is a critical need for many cloud services. For example, a cloud monitoring service needs to query thousands of hosts in a data-center to check for resource usage while a cloud homing service needs to find edge data centers across the world that satisfy certain complex constraints. This is a challenging problem, especially when confronted with highly dynamic state, scale on the order of hundreds or even thousands, geo-distribution and complex query constraints that traverse decentralized data sources. In this paper, we address this problem through the design of ***NodeFinder*** that is based on a novel pull-based approach in which we maintain decentralized (peer-to-peer) groups of nodes structured according to the node attribute values (i.e., their state). This allows queries to be sent to only a few representatives of the groups that have the potential of satisfying the constraints, and then the representatives gossip with their peers and return the latest set of nodes. This guarantees freshness of results, and ensures directed and thereby scalable querying. We show *NodeFinder*'s use in production use-cases such as host monitoring in our OpenStack clouds and NFV homing on edge clouds. Our preliminary experiments on Amazon EC2 illustrate *NodeFinder*'s scalability and efficiency as compared to today's approaches.

1 Introduction

In this paper, we introduce *NodeFinder*, a scalable service providing timely search across geo-distributed nodes with varied and highly dynamic state.

This is a critical service for cloud systems [3, 4, 7, 12], such as OpenStack, where they need to monitor the physical hosts' resource usage, such that admins and scheduling algorithms can identify nodes with certain properties (e.g., those that have high CPU utilization) in order to make decisions (e.g., migrate a virtual machine to another server). Edge cloud frameworks [10, 17, 21], as in ISP deployments of Network Function Virtualization (NFV) [16], make decisions on which network functions to run and where based on a variety of properties that are locally known, such as hardware capabilities and availability of host resources or the current profile of the traffic properties (e.g., to defend against a DDoS attack [19]). In each case, these applications need to incorporate some ad-hoc mechanism in order to find nodes (hosts or edge clouds).

This task has proven to be a challenging problem, and is becoming increasingly more challenging – mainly due to three intertwined reasons: (i) *Highly dynamic state*: the state of the nodes can change dynamically in the matter of minutes or seconds. For example, the resources available on a host (CPU, memory, disk) in a cloud system can change every few minutes based on the tenants running on the host, and similarly, the available bandwidth for a VM can change in the order of seconds - depending on the network activity. Hence, we need a system which provides search over fresh state. (ii) *Scale and geo-distribution*: the number of nodes can be on the order of thousands for the cloud and edge use-cases. Even worse, they could be distributed across the world (e.g., multisite clouds [11]) and are accessible through the wide-area network. At this scale, the resource requirements to enable search (in terms of bandwidth, compute power, etc.) cannot grow in direct proportion to the number of nodes in the system. (iii) *Complex queries spanning different data sources*: the queries for nodes in real-world services often contain constraints/sub-queries that need to be simultaneously satisfied by different nodes, and the information to answer these queries may be drawn from decentralized, heterogeneous data sources. For example, a single query may contain two sub-queries one of which requires node attributes from an OpenStack [12] cloud while the other requires AWS [3] cloud information (we elaborate more on this in §2.2).

As distributed applications continue to grow in size and geographical distribution, and as nodes are able (re-

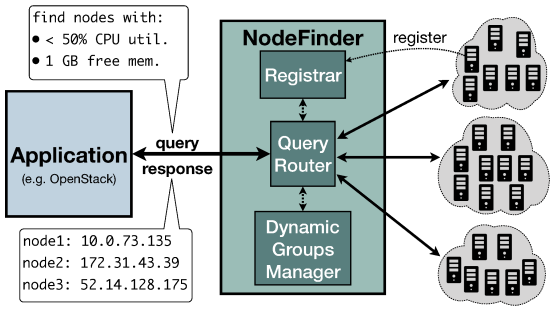

Figure 1: High-level overview of *NodeFinder* in which nodes are distributed across multiple cloud sites and form p2p groups (not shown) based on attribute values such as available CPU or memory for scalable query processing.

quested) to provide richer and more dynamic information, we are hitting the limits of existing approaches. Existing systems follow two main approaches: *push* or *pull*. In *push*-based approaches, the nodes periodically send updates to a centralized manager with their current state. While maintaining complete knowledge of every node in the network centrally makes the searching much easier, the excessive workload from perhaps thousands of nodes frequently updating their status in a highly dynamic environment renders the scheme very expensive. To cope, these systems tune the update frequency, but this, in turn, means searching over stale data. In *pull*-based approaches, when the application needs to find nodes, it will send a query to all nodes on-demand and request the current state. This approach provides the freshest information, but simultaneously (and frequently) querying such many nodes is not scalable (and may result in the TCP incast problem [22, 18]).

In *NodeFinder*, we address this trade-off through a hybrid architecture (Figure 1) wherein we intelligently group nodes based on attribute and value (i.e., their state). To realize this, nodes form peer-to-peer (p2p) groups with nodes which have similar values for a given attribute (e.g., average CPU utilization of 70% or more). These peers use gossip protocols [23] to exchange membership information internally and a representative of the group periodically pushes an aggregated update to *NodeFinder*, which then adjusts the groups based on this information. This means that when a query comes in, *NodeFinder* has already filtered nodes which *might* match the search (i.e., in groups that match the value, or in groups of nearby values). *NodeFinder* then only needs to send requests to those groups – providing timely data in a scalable manner. This goes beyond simply adding hierarchy, which only reduces a factor of the scale, and compromises the visibility that maybe required to answer complex global queries.

Simply put, *NodeFinder* fills a substantial need by providing a common service which can be integrated into any distributed system, solving a problem which has proven challenging for distributed systems. We have implemented a prototype of *NodeFinder* leveraging several systems [20, 6, 15] and deployed it on Amazon EC2, with nodes spread across 4 regions. Our evaluation shows that *NodeFinder* reduces bandwidth to the central server by 93% when compared to push/pull approaches, can perform queries at scale in less than 1 second, and does not impose overhead even for smaller scales.

2 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 *NodeFinder*: host monitoring in the cloud and virtual network function (VNF) homing on edge clouds.

2.1 Host Monitoring in the Cloud

A critical function in cloud management platforms is to monitor the physical hosts (e.g., compute nodes) that are hosting virtual machines. Such task involves tracking the inventory of VMs running on each host, their resource usage, tenant quota information, and other associated metrics. Typically, an agent running on each of the nodes updates the state of the corresponding host through a message broker. For instance, OpenStack clouds utilize RabbitMQ [14] as a messaging medium to convey resource status information. We discovered through extensive experiments on OpenStack that this approach does not scale well when increasing the number of compute nodes. The memory footprint of RabbitMQ increases linearly with an increase in compute nodes. This is primarily because Nova (the component in OpenStack which takes the messages off the message queues) reaches peak CPU utilization very fast and is unable to clear RabbitMQ queues. Our findings on scalability are backed up by results from the OpenStack community [13].

This scalability bottleneck is caused by every component in OpenStack frequently pushing its status update messages regardless of the need for them[1]. *NodeFinder* addresses this problem in a much more scalable and efficient manner. That is, each host runs a *NodeFinder* agent and reports its state as a set of attributes (e.g., number of VMs or CPU usage). *NodeFinder* then creates p2p groups based on these attributes such as: (nodes with 50 VMs or less), (nodes with 50 VMs or more), (nodes with 50% CPU utilization or more) and (nodes with 50% CPU utilization or less). *NodeFinder* maintains these groups and their representatives to whom the query can be sent

[1] except for healthchecks, where the updates are merely heartbeats.

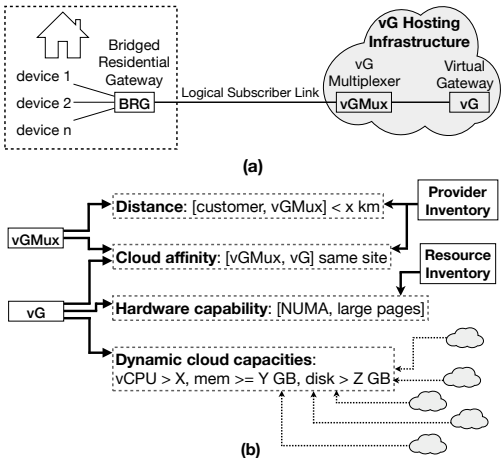

Figure 2: VNF homing: an apt use-case for *NodeFinder* that illustrates homing requirements for the residential vCPE service [8].

and then collect up to date state from their peers *only when needed*.

2.2 VNF Homing on Edge Clouds

A crucial network service management function for large scale Network Service Providers (NSPs) is the *homing service*, which uses optimization techniques to find suitable sites on which to deploy complex Network Services [9]. NSPs often have hundreds of sites (expected to increase to thousands for edge use-cases) spread across tens of countries hosting anywhere between 10 and 500 compute servers. The "home" (or location) of a VNF is chosen based on policies and requirements from service providers, cloud operators and VNF vendors. Homing is a challenging problem since the constraints that need to be satisfied for site selection often contain subqueries that needs to be simultaneously satisfied by different nodes, often requiring fresh information of dynamic node attributes (such as available CPUs on a host). Further, information to answer these queries may be drawn from decentralized, heterogeneous data sources (edge clouds such as Akraino [1], OpenStack clouds or even third party clouds like Azure [7] or AWS [3]).

Figure 2 shows the homing requirements of a real-world network service, virtual Customer Premises Equipment (vCPE) [8], that provides residential broadband connectivity. Figure 2(a) 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 vCPE located at the residential customer premises, while the vGMux is a shared network function at the PE that maps layer-2 traffic between a subscriber's BRG and its unique vG (which hosts the service related Network Functions), ensuring traffic isolation between multiple customers. Figure 2(b) shows the homing policies (or constraints) that drive the selection of an optimal PE site to host the vGMux and vG for a given customer. While constraints like *distance* or *cloud affinity* depend on relatively static cloud attributes (from service provider inventories), the other two constraints depend on attributes that are relatively more dynamic (from resource inventories and cloud providers). Resource capabilities within a cloud site may change as new host aggregates are added, while instantaneous site capacities may vary at even shorter time scales since resources are typically shared among multiple services and customers.

One of the principal challenges in *homing* is collecting and aggregating the information required for homing from multiple data sources in a holistic manner, and at different time granularities. Further, this information needs to be collected from thousands of sites including service provider-owned sites and third party clouds like Azure and AWS. The critical need is for a service that can provide a holistic, up-to-date list of the cloud sites that satisfy all the constraints. *NodeFinder* is well suited to address this problem, where each data source (including cloud sites) can simply run the *NodeFinder* agents forming p2p groups for the different attributes (e.g., one group for clouds that have CPU greater than a certain value while another group for clouds that have hosts with CPU pinning). Further, these agents can also act as "translators", using the cloud native APIs to acquire information about their attributes, which may differ across cloud providers. Instead, the homing service can simply query *NodeFinder* and use the aggregated information to identify sites that satisfy the constraints.

3 Abstractions

In this section, we provide a high-level overview of the abstractions provided by *NodeFinder*. Figure 1 depicts the high-level design of *NodeFinder* wherein an application can specify constraints for the nodes it wants to find, and *NodeFinder* will efficiently query the end nodes and return a list of nodes satisfying the constraints out of possibly thousands of nodes.

Node Attributes: Nodes have attributes that can be described as *static* or *dynamic*. Values of *static* attributes do not change (e.g., number of CPUs) while values of *dynamic* attributes can and do change over time (e.g., free memory).

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*, and a *freshness* parameter. The attribute *name* is used to describe the at-

tribute of interest to the requester application. The *upper bound* and *lower bound* values are used to support less/greater than operations. If an exact match is needed, then both bounds should be of the same value. 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 *NodeFinder* supports for other attribute types (e.g., location, text-based attributes, etc).

4 Under the Hood

NodeFinder (shown in Figure 1) consists of three main components elaborated here: a ***Registrar***, where nodes and their attributes are learned and maintained, a ***Groups Manager***, where nodes are placed into groups based on attribute values, and a ***Query Router***, where *NodeFinder* can selectively pull from a select group of nodes based on the filtering already performed in managing groups.

4.1 Node Registry

The very initial step for end nodes (running *NodeFinder* agents) is to register themselves with *NodeFinder* by consuming *NodeFinder*'s southbound API (the *Registrar*). The *Registrar* acts as a rendezvous point for newly arrived nodes and is responsible for knowing all nodes in the system. To facilitate the attribute based grouping, when a node registers, it will provide information about what attributes it has. *NodeFinder*, in turn, knows what and how many attributes it will be handling, which is needed to appropriately create and manage the corresponding p2p groups. In addition, upon node registration, *NodeFinder* classifies each of the provided attributes as *static* or *dynamic*, which is useful for query processing (*static* state is answered locally and *dynamic* state is fetched from nodes).

4.2 Dynamic Group Management

NodeFinder groups nodes based on their attribute values (one p2p group per range of attribute values) and dynamically adjusts these groups as nodes change values, in a loosely-coupled fashion. This is so that by the time a query comes in, *NodeFinder* has already pre-filtered nodes based on their attribute values to only those that have the potential to successfully match the query.

P2P Group Life-cycle: Upon node registration, and depending on its attributes and values, *NodeFinder* will suggest the appropriate p2p groups for each of the reported attributes, and the node then will join such groups. For the very first node that registers for an attribute, there will not be established groups yet, so *NodeFinder* instructs the node to start a group and act as a group representative. During the operation of a p2p group, a group representative pushes periodic updates (with a frequency set by *NodeFinder*) containing group membership information. *NodeFinder*, through its *Groups Manager*, uses this information to populate a mapping table to keep track of which nodes are in which p2p groups. The *Groups Manager* uses this information to decide to: start a "twin" group (e.g., when group size exceeds a threshold), terminate a group (e.g., empty group), or elect a new node as a group representative (e.g., when current representative times out). When *NodeFinder* refers a node to a group, it also announces the group boundaries (i.e., *upper* and *lower* bounds described in §3) so that the node leaves the group when the new values fall outside the specified range. Unless the new values fall outside the specified range of the group, nodes do not need to communicate with the *NodeFinder* server, thereby enabling a loosely-coupled style of node management.

4.3 Query Routing

NodeFinder adopts a *pull*-based approach to acquire dynamic node state. The use of groups enables *NodeFinder* to be highly scalable and efficient by significantly narrowing down the number of nodes to pull from. After determining the p2p groups based on adaptive attribute filtering (§4.2), the *Query Router* needs to route the query to the node to get the current value, and returns a fresh result to the query requester. As described in §3, each query request consists of a list of queryable attributes. For each of which, *NodeFinder* sends a query to the corresponding p2p group. When a member of the p2p group receives a query, it gossips the query with other nodes, and returns the aggregate results to *NodeFinder*.

Optimizations: We optimize the *Query Router* in three ways. First, our *Query Router* processes the list of attributes of a query request in parallel. Second, we cache results to serve future queries. Third, the *Query Router*, with the help of the *Groups Manager* (§4.2), sends the query only to the smallest p2p group in order to get faster responses. The latter is only done for queries whose constraints should be *all* met.

5 Evaluation

We have implemented a prototype of *NodeFinder* and used it to gain some preliminary insight into the scalability and efficiency of the queries. Our prototype leverages existing tools, such as: Apache Cassandra [20] as an inventory and Serf by HashiCorp [15] for the p2p groups. We deployed *NodeFinder* on Amazon EC2 [2], with nodes spanning 4 different regions in the United

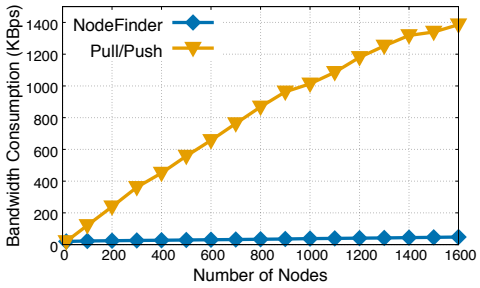

Figure 3: This preliminary experiment illustrates how *NodeFinder* scales much better with the number of nodes as compared to push/pull approaches, despite pulling information once every second to deal with the dynamic nature of the attributes.

States (400 nodes/region). Each node reported four attributes, and there were four p2p group bins per attribute resulting in 16 p2p groups in total with 100 nodes on average per group. For example, the four groups corresponding to percentage CPU utilization of a node were *CPU-25*, *CPU-50*, *CPU-75* and *CPU-100*.

Scalability of querying dynamic state: We measured the bandwidth consumption at the *NodeFinder*'s server (residing in one of the 4 regions) when querying nodes using three approaches: (i) *NodeFinder*'s approach where we pull information from the group representatives, (ii) a simple pull approach where the central *NodeFinder* server pulls information from each node, and (iii) a push approach where each node updates its information at the *NodeFinder* server. The querying frequency across the three approaches was set to once per second. Figure 3 shows that the bandwidth consumption of pull/push approaches [2] increases linearly as the number of nodes increases, while *NodeFinder*'s scale stays relatively constant. As a result, *NodeFinder* can eliminate about 93% of unneeded communication while maintaining service.

Resource overhead of running node agent: Our prototype of the *NodeFinder* node agent consumes about 10MB of memory for each p2p group it joins. Running *NodeFinder* agent at the group representative also does not introduce significant overhead, even when the representative queries the entire group every second. We found that when processing 1 query/second, it consumes about 5 to 50KBps for groups of 50 and 400 nodes, respectively. This also shows that *NodeFinder* does not impose significant overhead on small scale systems.

Query response time: *NodeFinder* demonstrated fast query processing when responding to queries targeting groups with 100, 300, and 500 nodes/group, where end-to-end query response times were 660*ms*, 900*ms*, and 945*ms*, respectively. These times can be further improved by tuning the p2p group configurations (fanout, frequency, etc) [5].

[2] pull and push both showed identical results.

6 Discussion and Future Work

Our work on *NodeFinder* is ongoing and we are exploring all aspects to enabling *NodeFinder* to serve as an essential service for existing and emerging cloud platforms and applications. First, while the current design of *NodeFinder* efficiently tackles the challenges of dynamic state, scale and geo-distribution, there is more research to efficiently tackle the challenge of complex queries spanning different data sources (as exemplified by our VNF homing use-case in §2.2).

Second, we plan on experimenting with different techniques for deciding the right splitting for the p2p group sizes. The current implementation assumes fixed and equally-split groups for each attribute. However, this might not always lead to the optimal decision as it could create imbalances across different groups for the same attribute. One way to resolve this is to use heuristics from real query traces as well as resource information heuristics from the end nodes to assist *NodeFinder* in deciding how to organize those dynamic p2p groups.

To enable wide-spread use of *NodeFinder*, we are also working toward integrating *NodeFinder* into existing cloud frameworks such as OpenStack [12] and network automation platforms like ONAP [10], especially with respect to the the use-cases described in this paper. Lastly, *NodeFinder*, as presented, is a system that responds to queries. Another highly related function is continuous monitoring for specific events (triggers). To do so, we can expand the node agents with triggering queries to be installed to offer a more automatic event monitoring mechanism.

7 Conclusion

We present *NodeFinder*, a scalable search service for highly dynamic and geo-distributed state by organizing similar nodes in p2p groups for faster and more scalable query processing. The design of *NodeFinder* is motivated by real world cloud use-cases. Our preliminary experiments with a prototype of *NodeFinder* suggest that it can be *scalable* (reduces resource usage of the search service by almost an order of magnitude), *lightweight* (does not impose significant overhead on end nodes), and *quick* (answers queries in less than 1 second).

Acknowledgement

This research was supported in part by the National Science Foundation under grants 1652698 (CAREER) and 1320389 (NeTS).

References

[1] Akraino Edge Stack. https://www.akraino.org/.

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

[3] Amazon Web Services (AWS) - Cloud Management Services. https://aws.amazon.com.

[4] Apache CloudStack. https://cloudstack.apache.org.

[5] Convergence Simulator - Serf by HashiCorp. https://www.serf.io/docs/internals/simulator.html.

[6] Eclipse Jetty. https://www.eclipse.org/jetty/.

[7] Microsoft Azure Cloud Computing Platform and Services. https://azure.microsoft.com.

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

[9] Onap homing and allocation service. https://wiki.onap.org/pages/viewpage.action?pageId=16005528.

[10] Open Network Automation Platform (ONAP). https://www.onap.org.

[11] Openstack cascading solution. https://wiki.openstack.org/wiki/OpenStack_cascading_solution.

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

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

[14] Rabbitmq. https://www.rabbitmq.com/.

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

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

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

[18] CHEN, Y., GRIFFITH, R., LIU, J., KATZ, R. H., AND JOSEPH, A. D. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proc. ACM Workshop on Research on Enterprise Networking (WREN)* (2009).

[19] FAYAZ, S. K., TOBIOKA, Y., SEKAR, V., AND BAILEY, M. Bohatei: Flexible and Elastic DDoS Defense. In *USENIX Security Symposium (USENIX Security)* (2015).

[20] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[21] PALKAR, S., LAN, C., HAN, S., JANG, K., PANDA, A., RATNASAMY, S., RIZZO, L., AND SHENKER, S. E2: A Framework for NFV Applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)* (2015).

[22] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proc. USENIX Conference on File and Storage Technologies (FAST)* (2008), FAST'08.

[23] SERUGENDO, G. D. M., GLEIZES, M.-P., AND KARAGEORGOS, A. *Self-organising software: From natural to artificial adaptation*. Springer Science & Business Media, 2011.