A High Performance, Scalable DNS Service for Very Large Scale Container Cloud Platforms

Haifeng Liu
JD.com Inc.
Beijing, China
University of Science and Technology of China
Heifei, China
bjliuhaifeng@jd.com

Shugang Chen
Yongcheng Bao, Wanli Yang
JD.com Inc.
Beijing, China
{chenshugang,baoyongcheng, yangwanli}@jd.com

Yuan Chen, Wei Ding
Huasong Shan
JD.com Silicon Valley R&D Center
{yuan.chen,wei.ding,huasong.shan}@jd.com

ABSTRACT

Containers and microservices are dominating the world of data center and cloud computing. As the scale, dynamism and complexity grow, the performance of the DNS system in container clusters becomes vital. As the world's third and China's largest e-commerce site by revenue, JD.com runs one of the world's largest Kubernetes container clusters in production. It is imperative that the DNS system can handle extremely high traffic. In this paper, we present ContainerDNS, a high performance DNS system for very large scale container clusters with millions of containers. ContainerDNS maximizes DNS system performance and scalability by optimizing DNS packet processing and using efficient memory and cache management.

ContainerDNS has been deployed in JD's container platform with 30,000 servers and 500,000 containers running tens of thousands of services and applications. It improves the maximum throughput from 130,000 to 9,000,000 QPS, a 67X performance boost comparing to existing DNS systems.

CCS CONCEPTS

• **Networks → Naming and addressing**; Cloud computing;

KEYWORDS

Container, Domain Name Systems, Cloud Computing, Performance, Scalability, Kubernetes

ACM Reference Format:
Haifeng Liu, Shugang Chen, Yongcheng Bao, Wanli Yang, and Yuan Chen, Wei Ding, Huasong Shan. 2018. A High Performance, Scalable DNS Service for Very Large Scale Container Cloud Platforms. In *19th International Middleware Conference Industry (Middleware '18 Industry), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3284028.3284034

1 INTRODUCTION

Cloud computing and middleware are quickly moving toward cloud-native approach that leverages containers and microservices [13–15, 22, 23, 25]. Applications are splitted into small microservices and deployed to a sea of containers [16]. A single application might consist of hundreds of services with thousands of instances, each running in a container such as Docker [2], and each service is typically associated with one or multiple virtual IPs as the service access point. Such a container platform usually employs the Domain Name Service (DNS) to manage the domain names of the services and their IPs.

For very large scale services whose instances are dynamically created and scheduled by an orchestrator like Kubernetes [6], It is a great challenge to provide a performant and scalable DNS system. For example, currently there are about 500,000 active containers hosting retailing related services in our data center. This extremely large scale container platform requires the DNS system to be able to handle very high volume of DNS lookup requests. The most recent data from our production system shows that the peak throughput to the DNS system can reach 3.5 million queries per second (QPS). Some public services such as monitoring service heavily rely on the DNS system, and can contribute much more throughputs than others. These services can constantly change due to service creation, deletion, update, elastic scaling, failure-recovery, etc. To the best of our knowledge, none of existing container-based DNS systems can be used in this large scale, high throughput, and dynamic production environment.

In addition, our evaluation shows that the existing container-based DNS systems could suffer great performance fluctuations due to dynamic memory allocation and garbage collection. This will severely hurt the performance of our online services and in turn affect the customer experience, especially when the throughput is high.

In this paper we present **ContainerDNS**, a container-based DNS system, to address these issues in our production environment. The key contributions of this paper are summarized as follows.

- We reveal the problems of the existing container-based DNS systems based on our past experience with JD's Kubernetes platform.
- We propose ContainerDNS, a scalable and high performance DNS system for very large scale container platforms to resolve these problems. The code is available at https://github.com/tiglabs/containerdns.

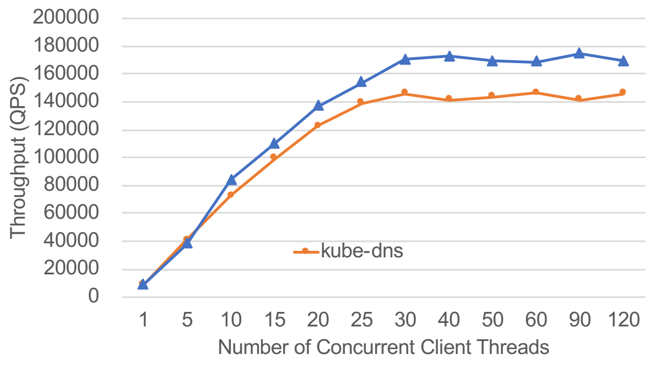


Figure 1: Scalability limitation in kube-dns and CoreDNS.

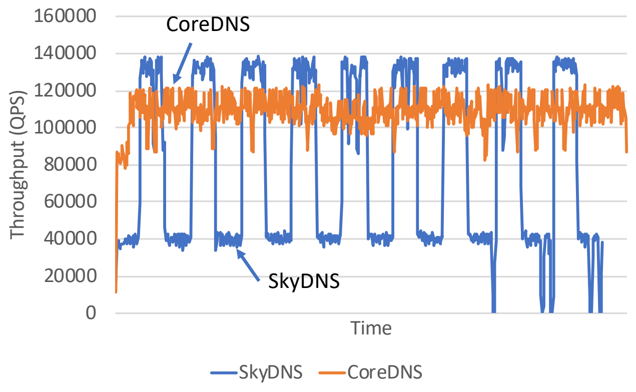


Figure 2: Performance fluctuation in SkyDNS and CoreDNS.

- We show that, in our experiments, ContainerDNS could achieve a maximum throughput of nearly 9 million QPS, about 67X better than that of the existing container-based DNS systems. It exhibits much more consistent performance with minimum variations comparing to existing systems, and its 99th percentile response time is less than 5ms at the peak throughput.

The rest of the paper is organized as follows. Section 2 explains the problems of the existing DNS systems on our Kubernetes platform. Section 3 describes the detailed design of ContainerDNS and related optimization techniques, and shows how the issues identified in Section 2 can be resolved by ContainerDNS in our production environment. Section 4 presents the experimental results with ContainerDNS and other existing DNS systems from various aspects. Section 5 reviews the related work and we conclude the paper with a summary and a description of future directions in Section 6.

2 PROBLEMS WITH THE EXISTING DNS SERVICES

kube-dns [5] and CoreDNS [1] are two popular DNS systems for Kubernetes container platforms. The kube-dns is integrated with Kubernetes as the default DNS solution. In the kube-dns, several containers are used and deployed together on the same host: *kubedns*, *dnsmasq*, and *sidecar*. The *kubedns* watches for changes in Services and Endpoints, and maintains in-memory lookup structures to serve DNS requests; the *dnsmasq* provides caching and stub domain support to improve the performance; and the *sidecar* provides metrics and health checks. The CoreDNS, a Cloud Native Computing Foundation incubating project, is the newest container-based DNS system. Compared to the kube-dns, the CoreDNS implements all the functionalities of the *kubedns*, *dnsmasq* and *sidecar* in a single container running a process written in Golang. The different plugins replicate and enhance the functionality found in the kube-dns.

As the business grows, during the process of launching more and more services/applications to JD's Kubernetes platform, several problems, such as scalability limitation and performance fluctuation, have been discovered in the existing container-based DNS systems.

2.1 Scalability Limitation

In order to examine the scalability, we use a varying number of concurrent client threads to send the DNS requests to each DNS system. The details of this experiment will be provided in Section 4. As can be seen from Figure 1, the maximum throughputs that the kube-dns and the CoreDNS can reach are 140,000 QPS and 170,000 QPS, respectively. However, in our production environment, the throughput of DNS requests can be as high as 3.5 million QPS, which means that neither kube-dns nor CoreDNS could meet our needs.

2.2 Performance Fluctuation

In order to examine the stability, we pre-populate 2 million DNS records[1] to the DNS system in order to simulate the production environment. We are not able to perform this experiment for kube-dns,[2] but we include the results with SkyDNS [9], a well-known DNS component that was used in the previous version of kube-dns, to show that the performance fluctuation is a common issue in existing DNS systems. As can be seen from Figure 2, the maximum throughput of SkyDNS varies from 40,000 QPS to 140,000 QPS (after omitting the noise). CoreDNS suffers a similar issue and the maximum throughput varies between 80,000 QPS and 120,000 QPS (after omitting the noise). A further profiling analysis reveals the performance variations are mainly caused by dynamic memory allocation and garbage collection arising from the implementations in Golang.

3 THE CONTAINERDNS

Figure 3 illustrates the high level architecture of ContainerDNS, which has four main components: *DNS server* (with a *DNS cache*), *DNS datastore*, *Service monitor*, and *IP status probe*.

The *DNS server* is the brain of ContainerDNS that serves for all the incoming DNS requests. The *DNS datastore* is a distributed

[1]A DNS record is a data structure that consists of a domain name, corresponding IP addresses, and other related fields.
[2]A common way to perform this experiment is to inject a large amount of DNS records to the persistent storage (e.g., a key-value store) directly connected to the DNS system. However, the latest version of kube-dns does not directly interact with such a persistent storage anymore, which make it very difficult for us to perform this experiment in a similar way.

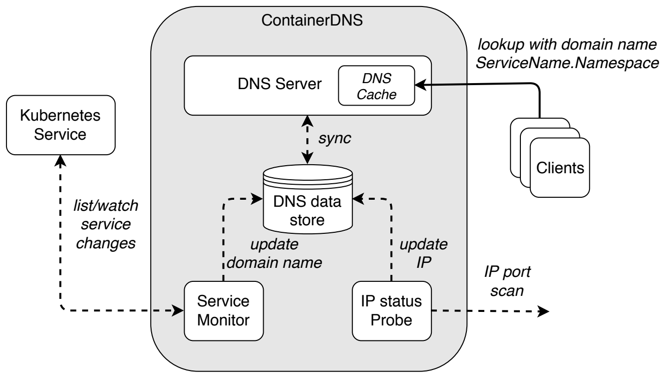


Figure 3: ContainerDNS architecture.

key/value store that persists all the DNS records. The *DNS cache* resides on the server and has a copy of the data (up to its cache size limit) in the datastore. Upon receiving a lookup request from the client, the DNS server will first try to fetch the DNS record from the cache directly. If this operation fails, then the server will either return with an error message (for local lookup request[3]) or forward the request to other DNS server (for non-local lookup request). In addition, whenever there is an update in the datastore, the server will be notified to synchronize its cache, so that the client has less chance to obtain the stale data.

The *Service monitor* watches for service and endpoint changes in real time and updates the DNS datastore accordingly. It provides both incremental synchronization and full synchronization. The former one keeps the datastore up-to-date for any service or endpoint change, and the latter one periodically (every 10 minutes by default) synchronizes the full data set of the service and endpoint information with that in the DNS datastore to ensure the data consistency. The full synchronization is necessary since the incremental synchronization may fail due to the network failure, etc.

The *IP status probe* keeps the DNS datastore up-to-date for any IP change in real time. It consists of a probe-scheduler and multiple IP-scanners. Whenever there is a service or endpoint change (notified by the Service monitor), the probe-scheduler will assign a probe task to one of the IP-scanners, who will then actively scan the IP addresses and ports for liveness, and update the DNS datastore accordingly.

It should be noted that, there can be multiple instances of the Service monitor and the IP status probe for performance and availability. As a result, a change in the container cluster can lead to concurrent updates in the DNS datastore from multiple monitor/probe instances. To address this issue, the monitor/probe first performs a check in the datastore, and aborts the write operation if the data has already been updated.

ContainerDNS has been fully integrated with Kubernetes to provide better service management for large scale container clusters, including hot upgrade and autoscaling in multiple domains. A network routing process such as Quagga [8] has been deployed on each

[3]For example, if a DNS server is responsible for serving all the queries to *.tst.local, then the request for resolving *.tst.local is a *local lookup request*.

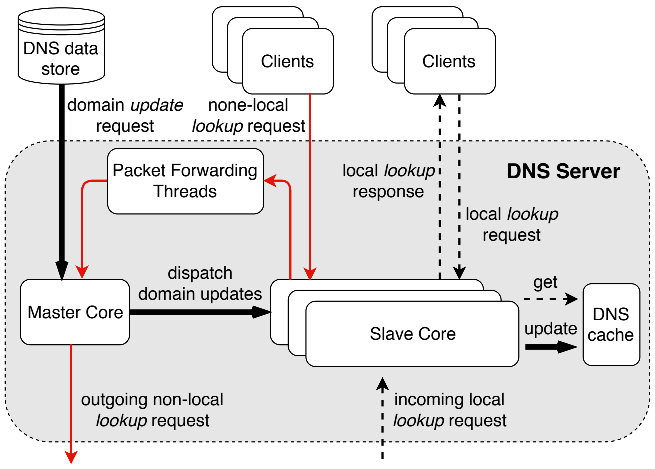


Figure 4: Internal structure of DPDK-based DNS server.

physical machine running ContainerDNS. Shutting down the routing process on a machine will disable the corresponding DNS server. In this way, it is convenient to perform dynamic rollbacks (to handle failures) and upgrades without service interruption. Similarly, by starting or killing the routing processes, we can dynamically scale in/out the number of machine running ContainerDNS. This design enables ContainerDNS to achieve hot upgrade, high availability and scalability in a simple manner.

Next we discuss the key techniques used in ContainerDNS to serve high throughput and very large scale container platforms.

3.1 Improving Performance/Scalability by Accelerating Packet Processing

Our profiling results show that both kube-dns and CoreDNS spend a lot of time on sending, receiving and forwarding the DNS packets on the server side due to the involvement of the Linux protocol stack for every packet. Motivated by this, the *DNS server* in ContainerDNS is designed with the goals of minimizing the overhead of processing the packets and maximizing the concurrency of operations, hence achieving the maximum throughput.

Figure 4 gives the internal structure of the DNS server, which leverages DPDK [3], a set of libraries and drivers written in C, to develop fast packet processing with minimum CPU cycles. The DNS server consists of one master core and multiple slave cores. The master core is mainly used to (1) dispatch the update requests of the domain names (from the data store) to the slaves, and (2) forward the lookup requests of non-local domain names (originated from the clients) to the network. The slave core is responsible for processing the update requests from the master core, and handling the lookup requests of local domain names.

Specifically, when a domain has been updated in the data store, an update request will be sent to the master core, who will then assign one of the slaves to update the cache (see the thick arrow lines in Figure 4). On the other hand, upon receiving a lookup request over the network, the slave will first check if the domain name is in the local zone or not. If the answer is yes, then the slave will just return the data fetched from the DNS cache to the client (see the dash arrow lines in Figure 4). Otherwise, it will forward the request to a set of background threads for future analysis and process. The processed request will be sent back to the master core, who will then send it out to the network through the Receive Side Scaling (RSS) to the upper level DNS server for resolution[4] (see the red arrow lines in Figure 4).

There are three optimization techniques used for packet processing, as explained below:

- **Lock-free sending and receiving queues**. Each slave core processes data on a receiving queue and sends the packets back to the master core through a sending queue. Therefore, each slave core independently sends and receives packets without requiring any locking mechanism, improving the concurrency and performance of packet processing.
- **Background threads to process non-local requests**. Because processing the non-local domain names can slow down the local zone name resolution, we use a set of background threads for packet forwarding and processing. These threads are not bound to any CPU cores and are completely separated from the master and slave cores. When a slave core receives a non-local domain name, it places the request into a queue shared among the background threads. Thus, the slave cores can resolve the local domain name quickly without being blocked. A background thread reads request from the queue, pre-processes the data and forwards it to the upper level DNS server. It then adds the result to the queue shared with the master core. By employing the background threads, we remove the work of dealing with non-local domain names from the critical path of processing the local domain names. Therefore, the concurrency and performance of packet processing are able to be further improved.
- **Shared memory buffer**. Since the data is shared between the background thread and the master core thread via shared memory buffer, there is no data copy involved during the entire operation, which further improves the performance of processing the packets.

3.2 Improving Performance by Effective Cache Management

Most of the existing DNS systems use TTL-based cache replacement policy. There are two problems associated with this approach. First, the cache performance is very sensitive to the cache size. A DNS system with good performance typically requires a large cache size, which can be expensive. Second, in a dynamic container environment, the domain names can be updated frequently and data consistency and freshness can become a problem.

To address these issues, we propose to actively monitor the change of the domain names and update the cache accordingly. In this way, the cached data never ages and the consistency between the cached data and the original data in the datastore gets improved (i.e., the possibility of accessing stale data from the cache is reduced). When the cache is full, the DNS server randomly selects the exceeded number of data records and deletes them from the cache.

[4]Receive side scaling (RSS) is a network driver technology that enables the efficient distribution of network receive processing across multiple CPUs in multiprocessor systems.

Table 1: Measured resource usages of ContainerDNS with kube-dns and CoreDNS at the maximum throughput as shown in Figure 5(c).

Systems	Max QPS	CPU (%)	Mem.(%)	Net.(MB/s)
kube-dns	131,911	24.5	4.3	25.8
CoreDNS	174,738	10.1	4.4	37.4
ContainerDNS	8,800,085	16.3	4.6	1729.5

This real time monitoring/updating mechanism work wells under normal conditions, however, it may fail to detect the changes in the datastore due to machine or network failures, which could happen several times a week in our system, causing data inconsistency between the cache and the datastore. In order to prevent this happening, a full synchronization between the cache and the datastore is performed periodically. To minimize the overhead, the DNS server records the most recent update time for each domain name, and if a change is more recent than the time of the previous synchronization, there is no need for update.

3.3 Minimizing Performance Fluctuation by Efficient Memory Management

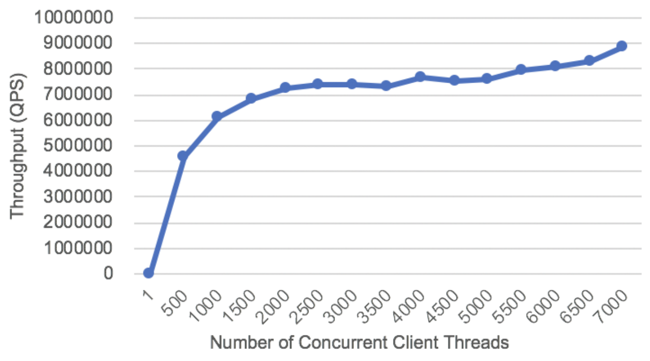
Existing container-based DNS systems such as kube-dns and CoreDNS are implemented in Golang and their performance varies a lot due to memory allocation and garbage collection. This has a huge impact on the application performance and availability when the traffic to the DNS system is high, especially for the latency-sensitive online services. For example, as shown in Section 2.2, the peak throughput of SkyDNS varies a lot from 40,000 QPS to 140,000 QPS. The analysis of the profiled data reveals that it dynamically allocates 200GB memory and the memory garbage collection later leads to the performance fluctuation. kube-dns and CoreDNS suffer a similar issue though to a less severe extent.

Because our DPDK-based DNS server is implemented in C, we are able to pre-allocate memory blocks on the collected memory usage and explicitly manage the memory pool on demand. These fixed-size memory blocks can only be accessed through a lock-free queue, which greatly reduces the need for dynamic memory allocation, and therefore minimizes the performance jitter.

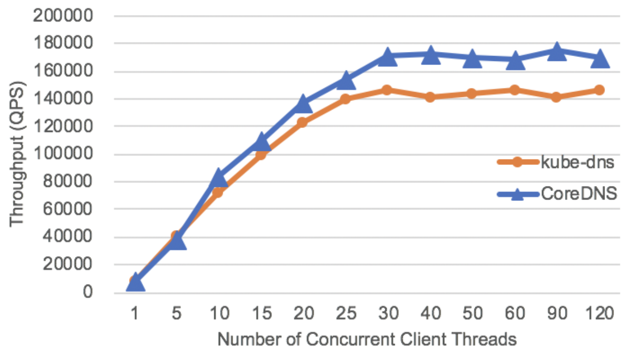
4 EVALUATION

In this section, we will present the experimental results with ContainerDNS and other existing DNS systems from various performance and scalability aspects.

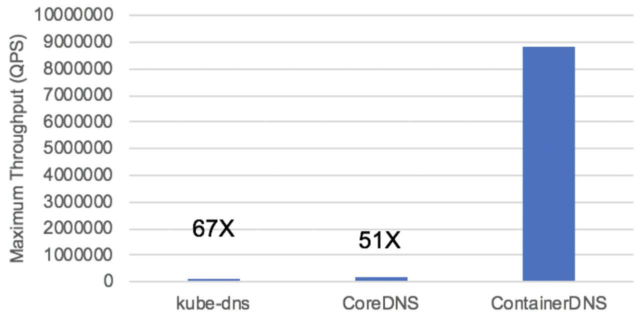
In our experiment setup, each instance of DNS system (e.g., kube-dns, CoreDNS, and ContainerDNS) runs on a set of machines with Intel® Xeon® CPU E5-2698 v4@ 2.20GHz (80 cores), 256GB DRAM and 82599ES-10-Gigabit SFI/SFP+ network connection, and a certain number of DNS records are pre-populated to each DNS system. Concurrent client threads continuously send random DNS requests to the DNS server using JMeter [4]. Unless specified, each run lasts 15 minutes and the performance is measured every second using queryperf [7].



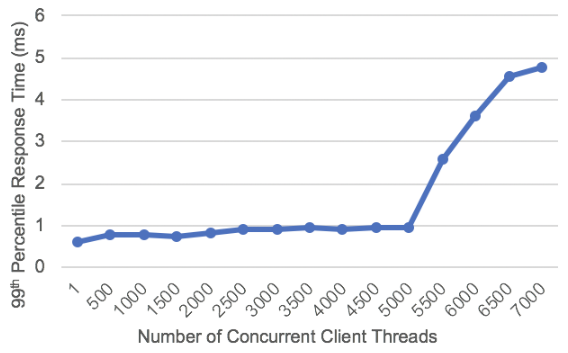
(a) Scalability of ContainerDNS.



(b) Scalability of kube-dns and CoreDNS.



(c) Comparison of maximum throughput.



(d) 99th percentile response time of ContainerDNS.

Figure 5: Comparison of scalability and latency.

4.1 Throughput and Latency

To compare the throughput, we pre-populate 50,000 DNS records to each DNS system, and use varies number of concurrent client threads to measure the throughput. Figure 5(a) and (b) shows the maximum throughputs of ContainerDNS and its competitors as the number of concurrent client threads increases.[5] It can be seen that, the throughput of ContainerDNS keeps increasing to about 9 million QPS with 7700 concurrent client threads. In comparison, kube-dns and CoreDNS can only reach the maximum throughput of 140,000 QPS and 174,000 QPS respectively with about 30 concurrent client threads. The maximum throughput of containerDNS is 67X and 51X better than that of kube-dns and CoreDNS, respectively (see Figure 5(c)).

We further compare the resource usages of ContainerDNS with kube-dns and CoreDNS at the maximum throughput. The results are shown in Table 1. ContainerDNS consumes the similar amount of CPU and memory resources as that of the other systems despite achieving a much higher throughput. The network bandwidth usage of ContainerDNS is much higher than the others because of its extremely high request rate and throughput as expected.

In addition Figure 5(d) plots the 99th percentile response time for ContainerDNS during this experiment. The latency is less than 1ms with 5000 concurrent client threads or fewer, and less than 5ms at the peak throughput with 7700 concurrent client threads. In our production environment, we have not met any issue with the latency in this range.

4.2 Performance Stability

To exam the performance stability, we pre-populate the same set of 2 million DNS records to each DNS system and use 30 concurrent clients to send out the DNS requests.[6] Figure 6 illustrates the throughput stability of ContainerDNS comparing to SkyDNS and CoreDNS.[7]. As can be seen, ContainerDNS has the least performance fluctuation with the highest throughput. Its throughput only varies between 180,003 QPS and 187,293 QPS. By contrast, the throughputs of SkyDNS and CoresDNS fall into the ranges of [40,000 - 138,000] QPS and [78,000 - 123,000] QPS, respectively (after omitting the noise).

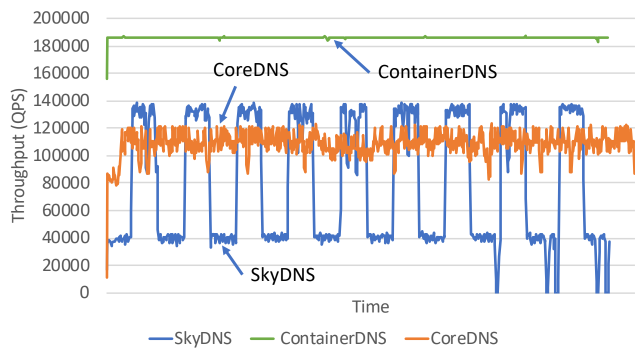
4.3 Performance Impact on Cache Size

We evaluate the impact of different cache sizes by comparing the performance of CoreDNS (which uses the TTL-based cache replacement policy) with ContainerDNS (which monitors the change in

[5]Figure 5(b) is copied from Figure 1 in Section 2.1. Putting it here is just for the purpose of better illustrating the scalability advantage of ContainerDNS.

[6]30 concurrent clients are used for the purpose of comparison. ContainerDNS can achieve much higher concurrency and throughput with many more clients.

[7]Please refer to Section 2.2 for the reason of choosing SkyDNS instead of kube-dns.



Systems	Average	Median	Max.	Min.
SkyDNS	81,083	43,224	138,564	1
CoreDNS	108,822	109,638	123,155	78,428
ContainerDNS	186,003	186,011	187,293	184,152

Figure 6: Throughput stability compared against SkyDNS and CoreDNS.

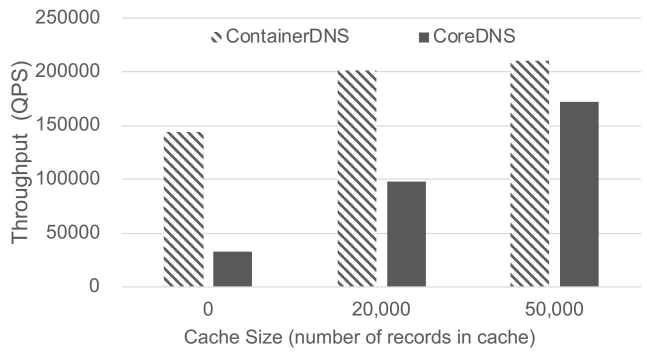


Figure 7: Performance impact on different cache sizes compared against CoreDNS.

the datastore and updates the cache in real time). We were not able to obtain the result from kube-dns due to its native integration with Kubernetes. In this experiment, we pre-populate 50,000 DNS records to each DNS system, and use three cache sizes (in terms of the number of domain names): 0 (no cache), 20,000 (partial cache) and 50,000 (full cache) with 120 concurrent client threads to send the DNS requests.

Figure 7 shows that ContainerDNS exhibits a much better throughput than CoreDNS (when with the same cache size). By contrast, CoreDNS's TTL-based method (TTL is set to 60 seconds) requires a very large cache size to achieve similar throughput. In addition to the performance advantage, ContainerDNS provides near real time cache update and almost eliminates the possibility of getting outdated data. In an experiment that lasts for a few weeks with addition, deletion, update and lookup operations, the detection of a change is generally within 20ms where the worst case is no more than 60ms, and billions of operations never returned any inconsistent or stale records.

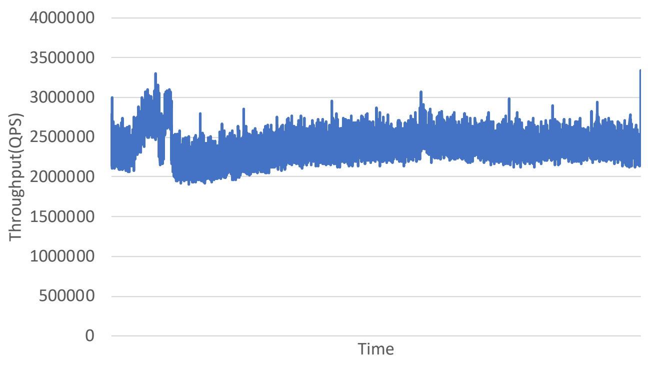


Figure 8: Throughput of ContainerDNS on July 31, 2018 at JD.com.

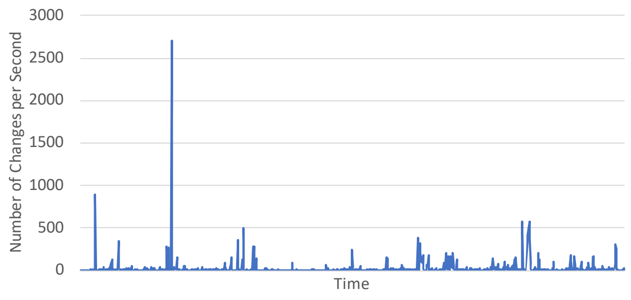


Figure 9: Changes of DNS records on July 31, 2018 at JD.com.

4.4 Real Production Data at JD.com

ContainerDNS has been deployed on one of the largest production container clusters in the world with 30,000 high end servers, 500,000 containers hosting 40,000 microservices and serving hundreds of billions online transactions every day. The container clusters continue to increase at a pace of 10,000 containers and 500 services every day. Figure 8 shows the throughput of ContainerDNS on July 31, 2018 at JD.com. The average rate is 2.4 million QPS and the peak reaches 3.5 million QPS.

Like typical microservice applications, the DNS records in our clusters varies dramatically as the services/applications constantly change because of service creation, update, elastic scaling and failure-recovery, etc. Figure 9 illustrates the number of domain-name changes on July 31, 2018. There were totally 24,772 domain-name changes within a 9-hour period of time on that day, and the maximum change rate is 2800 domain-name changes per minute.

5 RELATED WORK

DNS related systems have been studied extensively in both industry and academia since Morckapetris and Postel invented the Internet DNS and proposed the first DNS architecture in 1980's [21]. An active topic is DNS performance measurement [18, 26]. Load balancing is another area that has been investigated substantially [11, 17, 24]. As the core service of Internet and web applications, the security of DNS is critical. A fair amount of work has been done to address the DNS security issue [12, 20, 27]. All of the prior work has focused on the Internet domain name systems. None of them has addressed the DNS system in container clusters running a large number of dynamic micro services.

Existing container platforms use DNS software such as SkyDNS [9], kube-dns [5] and CoreDNS [1]. As discussed before, their performance is not sufficient when used in vary large clusters with millions containers like what we have in our data centers. Bind9 is a general DNS protocol and system. It was not designed for container platforms and the support of service discovery and other features are missing. Compared with the above systems, ContainerDNS offers not only the best performance but also the full integration and support of container and microservice management on Kubernetes cluster platforms.

6 CONCLUSION AND FUTURE WORK

New challenges arise with the coming era of cloud native computing with microservices running in container and managed by orchestration engines like Kubernetes [6], Mesos [19], and Swarm [10]. As the scale and complexity of container clusters is growing, exiting DNS system software cannot meet the performance and scalability requirements. We propose ContainerDNS and demonstrate that the use of advanced packet processing, optimized concurrency and efficient memory management can significantly improve DNS system performance and scalability. To the best of our knowledge, it is the first DPDK-based DNS system for container platforms that can support up to 9 million requests per second with a response time less than 5ms. It also offers better cache performance, data consistency and service management. ContainerDNS has been deployed on one of the world's largest Kubernetes clusters in production at JD.com.

REFERENCES

[1] [n. d.]. *CoreDNS*. "https://coredns.io/".
[2] [n. d.]. *Docker*. "https://www.docker.com/".
[3] [n. d.]. *DPDK*. "https://dpdk.org/".
[4] [n. d.]. *Jmeter*. "https://github.com/apache/jmeter".
[5] [n. d.]. *Kube-dns*. "https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/dns".
[6] [n. d.]. *Kubernetes*. "https://kubernetes.io/".
[7] [n. d.]. *perfquery*. "https://linux.die.net/man/8/perfquery".
[8] [n. d.]. *Qugga*. "https://www.quagga.net/".
[9] [n. d.]. *SkyDNS*. "https://github.com/skynetservices/skydns".
[10] [n. d.]. *Swarm*. "https://docs.docker.com/engine/swarm/".
[11] Charles Edward Anderson IV, Thomas Carroll Willis Jr, and Jason Andrew Willis. 2006. System, method and computer program product for caching domain name system information on a network gateway. US Patent 7,152,118.
[12] Roy Arends, Rob Austein, Matt Larson, Dan Massey, and Scott Rose. 2005. *DNS security introduction and requirements*. Technical Report.
[13] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2015. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 201–215.
[14] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software* 33, 3 (2016), 42–52.
[15] David Bernstein. 2014. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing* 3 (2014), 81–84.
[16] Eric A Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 167–167.
[17] Thomas Brisco. 1995. DNS support for load balancing. (1995).
[18] Peter B Danzig, Katia Obraczka, and Anant Kumar. 1992. An analysis of wide-area name server traffic: a study of the Internet Domain Name System. *ACM SIGCOMM Computer Communication Review* 22, 4 (1992), 281–292.
[19] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.
[20] Bill Karakostas. 2013. A DNS architecture for the internet of things: A case study in transport logistics. *Procedia Computer Science* 19 (2013), 594–601.
[21] Paul Mockapetris and Kevin J Dunlap. 1988. *Development of the domain name system*. Vol. 18. ACM.
[22] Claus Pahl. 2015. Containerization and the paas cloud. *IEEE Cloud Computing* 2, 3 (2015), 24–31.
[23] Robert Sandoval et al. 2015. *A case study in enabling DevOps using Docker*. Ph.D. Dissertation.
[24] Eric Sven-Johan Swildens, Richard David Day, and Vikas Garg. 2004. Scalable domain name system with persistence and load balancing. US Patent 6,754,706.
[25] Johannes Thönes. 2015. Microservices. *IEEE software* 32, 1 (2015), 116–116.
[26] Roland van Rijswijk-Deij, Mattijs Jonker, Anna Sperotto, and Aiko Pras. 2016. A High-Performance, Scalable Infrastructure for Large-Scale Active DNS Measurements. *IEEE Journal on Selected Areas in Communications* 34, 6 (2016), 1877–1888.
[27] Brian Wellington. 2000. *Secure domain name system (DNS) dynamic update*. Technical Report.