2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops

Secure Cache Provision: Provable DDOS Prevention for Randomly Partitioned Services with Replication

Weibo Chu*, Xiaohong Guan*†, John C.S. Lui‡, Zhongmin Cai* and Xiaohong Shi§
*MOE KLINNS Lab, Xi'an Jiaotong University, Xi'an, China
†Center for Intelligent and Networked System and NLIST Lab, Tsinghua University, Beijing, China
‡Department of Computer Science & Engineering, The Chinese University of Hongkong, Hongkong
§Beijing Qihoo Technology Co. Ltd, Beijing, China
Email: {wbchu, xhguan, zmcai}@sei.xjtu.edu.cn, cslui@cse.cuhk.edu.hk, shixiaohong@360.cn

***Abstract*—In this paper we show a small but fast popularity-based front-end cache can provide provable DDOS prevention for randomly partitioned cluster services with replication. To achieve this, we first give the best strategy for an adversary to overload the system, and then prove that the cache size is lower bounded by $O(n \log\log n / \log d)$, where n is the number of back-end nodes and d is the replication factor. Since $\log\log n / \log d < 2$ holds for almost all the current clusters (i.e., the number of back-end nodes $n < 10^5$ and the replication factor $d \geq 3$), this result implies an $O(n)$ lower bound on the required cache size. Our analysis and results are well validated through extensive simulations.**

I. Introduction

Today's cloud computing infrastructures employ back-end nodes at the scale of thousands or even larger [1]-[3] to provide sustainable services. For example, Google's BigTable and GFS cells have 1000-7000 nodes in one cluster [4], Facebook's photo storage has 20 petabytes of data [5], Microsoft's data mining cluster has 1800 nodes [6], and Yahoo's Hammer cluster has 3800 nodes [7]. For all these cloud service providers, protecting their large-scale clusters from various attacks such as DDOS (denial-of-service) attack is always a critical task.

To ensure performance scalability for clusters, many approaches [8]-[10] have been proposed. Load balancing among multiple back-end nodes using a front-end cache is perhaps the most widely adopted solution to provide sustainable services. In this cluster architecture as shown in Figure 1, *a small and fast popularity-based front-end* cache is employed to directly serve a very small amount of popular items and leaves uncached items to the back-end nodes. The front-end cache thus serves like a filter that regulates the skewed workload and makes the load across back-end nodes more uniform. Meanwhile, this front-end cache is generally small enough to fit in the L3 cache of a fast CPU, enabling an efficient and high-speed implementation of load balancer.

Besides employment of front-end cache, *replication* [11] is another technique widely adopted in today's clusters to avoid performance bottlenecks and ensure service-level objectives (SLOs). Replication refers to having multiple back-end nodes (also called replica group) to serve a single service request, where each request is served by one of the corresponding servers chosen according to some rules (i.e., random selection or in a round-robin fashion). With replication, cluster systems can provide substantial performance enhancement as well as the needed capacity of dealing with unpredictable performance variations that due to a number of factors (non-deterministic thread scheduling, hardware differences, network conditions, etc). Meanwhile, fault tolerance and reliability of the system is also greatly enhanced.

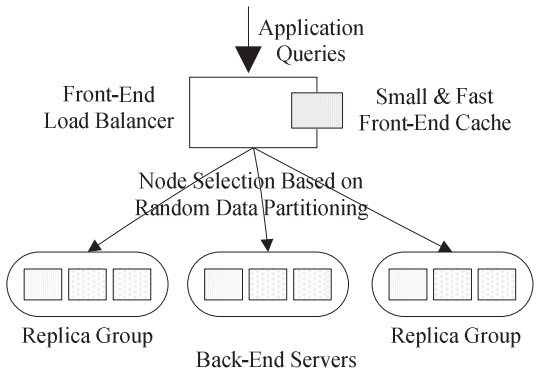


Fig. 1. Cluster with a small fast front-end cache and replication

In this paper, we investigate the effectiveness of the front-end cache size on providing provable DDOS prevention for randomly partitioned cluster systems with replication. More specifically, we aim to answer the following question: *how large the front-end cache should be for the cluster systems to be protected from DDOS attacks, i.e., that caused by adversarial access patterns?*

The above question arises naturally for large-scale cluster designers and managers, where the two techniques—load balancing using front-end caches and replication among back-end nodes, are simultaneously deployed in their systems. To answer this question, we begin by examining the best strategy (i.e., the most adversarial access pattern) that an adversary could adopt in order to overload the system, and

Corresponding author: Zhongmin Cai. The research is supported by NFSC (61221063, 61175039, 60905018), 863 High Tech Development Plan (2007AA01Z464, 2012AA011003), Research Fund for Doctoral Program of Higher Education of China (20090201120032), 03 Project "China New Generation of Broadband Wireless Mobile Communication Network" (2012ZX03002001) and Fundamental Research Funds for Central Universities (xjj20100051, 2012jdhz08).

978-0-7695-5023-7/13 $25.00 © 2013 IEEE
DOI 10.1109/ICDCSW.2013.24

then prove that the front-end cache needs to only store $O(n \log\log n / \log d)$ entries to provide provable DDOS prevention, where n is the total number of back-end nodes, d is the replication factor for each entry. This result has a significant meaning for today's cloud service providers in that: 1) the required cache size only depends on the number of back-end nodes and the replication factor, and not on the number of items served by the system, which means the front-end cache is scalable to the items served; and 2) as $\log\log n / \log d < 2$ holds for almost all the current clusters (i.e., the number of back-end nodes $n < 10^5$ and the replication factor $d \geq 3$), it implies an $O(n)$ lower bound on the required cache size.

Our work is an extension to [18], where the authors focus on providing provable load balancing for randomly partitioned services using a small fast front-end cache. In this work we extend the analysis to services with replication which is often the case in today's cloud computing infrastructures. With replication, we show that the results (the best strategy for the adversary, the workload on system, and the bound on the cache size) are quite different. One of the most important results we have is that when the cache size exceeds a certain threshold (our bound), the workload of the most loaded nodes in cluster will never be greater than the average load on system (i.e., when the workload is distributed evenly on back-end nodes). This result actually implies a provable DDOS prevention for the system through proper cache provision.

The rest of the paper is organized as follows. Section II presents our system settings and assumptions. Section III analyzes the most adversarial access pattern that an adversary could adopt to overload the system, and derives the bound on the cache size for providing provable DDOS prevention under this worst case. Section IV presents our simulation studies. We conclude the paper in Section V.

II. System Settings and Assumptions

A. System Settings

Throughout the paper we consider services (systems) with the following four properties. These services are now critical parts of several major Internet services.

1) *Randomized partitioning*. The service is partitioned across back-end nodes and the way the service is partitioned is opaque to the clients (i.e., a key is hashed to select the back-end nodes that serve it).
2) *Equal replication*. The replication factor for each item is equal (i.e., the number of back-end nodes that can serve any two different keys is the same). And the rule used to map each item to the node (one of the replica nodes) which ultimately serves it is the same (i.e., through randomly choosing or in a round-robin fashion).
3) *Cheap to cache result*. The front-end cache can easily store the result of a query or request and can serve future request without recomputing or retrieval.
4) *Costly to shift results*. Moving service from one back-end node to another is expensive in network bandwidth, I/O, consistency, etc. In other words, the partitioning is relatively stable on the timescale of a few requests.

Systems that fall into this category include:

- **Distributed file systems** such as Google File System (GFS) [12] or the Hadoop Distributed File System (HDFS) [2], where each data block is located and served at one or multiple semi-random servers.
- **Distributed object caches** such as memcached [13].
- **Distributed key-value storage systems** such as Dynamo [14], Haystack [5], and FAWN-KV [15].

Services we *do not consider* in this work include:

- **Queries can be handled by any node**, such as a web farm with a large set of identical servers, each of which is capable of handling any request. These services do not need caching for effective load balancing.
- **Partitioning is predictable or correlated**. For example, column stores such as BigTable [17] and HBASE [16] store lexicographically close keys on the same server. For these systems, an attacker could potentially forward a large set of keys to the same back-end node or replica group, and our results apply only when keys are partitioned independently.

B. Assumptions

We make the following assumptions in our analysis:

1) *Randomized mapping*. Both the mapping of each key to its replica group and the mapping of the key to the node ultimately serves it are randomized, and are unknown to the client (adversary). Although the node in each replica group which ultimately serves the query is often selected based on some deterministic rules, i.e., always choosing the least loaded node, the mapping of the keys to their replica groups is unknown (and hence randomized) to the adversary. Therefore it is reasonable to assume that the mapping of the key to the node that ultimately serves it is also unknown and randomized.

2) *Perfect caching*. The front-end cache can always cache the most popular items. Queries for these items could always hit the cache while other items always miss the cache.

3) *The cache is fast enough*. As compared to the back-end nodes, the front-end cache is fast enough to handle queries and never becomes the bottleneck of the system.

4) *Uniform cost*. The cost to handle a query at the back-end node is the same, regardless of the type of queries or which node processing the query.

Note that real-world clusters can be much more complicated, for example, there can be multiple types of operations (read, write, update, etc) and some can cost more resources. In [18], the authors discussed how to address these issues. We believe their techniques are also applicable to our analysis as our system settings and assumptions are almost the same.

III. Analysis

This section is devised to analyzing the workload on cluster system. We will first analyze the best strategy (i.e., the most adversarial access pattern) that an attacker could adopt to overload the system, and then derive the bound on the required cache size for providing provable DDOS prevention for the system under this worst access pattern.

TABLE I
NOTATIONS IN ANALYSIS

Symbol	Meaning
n	# of back-end nodes
m	# of (key, value) items stored in the system
c	# of (key, value) items cached in the front
d	# of back-end nodes that can serve an item, also replica factor
R	sustainable query rate
L_i	query rate going to node i
r_i	max query rate supported by node i
p_i	fraction of queries for the ith (key, value)

A. Adversarial Access Pattern

System Model Consider a system such as that shown in Fig. 1 which serves a total of m district items partitioned uniformly across n back-end nodes. Each item can be served by d corresponding back-end nodes (replica group), and once queried, the item is served by one of these d nodes chosen according to some rules (i.e., random selection or in a round-robin fashion). Each node i ($1 \leq i \leq n$) can handle at most r_i queries per second. The system caches the c most popular items ($c \leq m$) at the front-end. On a cache hit, the front-end can serve the client request without querying the corresponding back-end nodes. Table 1 summarizes the notation used in our analysis.

In order to generate skewed workload to overload the system, we assume the adversary knows which m keys are stored on the system, the number of back-end nodes n, the replication factor d and the front-end cache size c.

However, the adversary doesn't know which keys are assigned to which replica group or nodes, so it is not easy for the adversary to target a specific group or node as the hotspot.

What the adversary can do is to launch queries for certain amount of different keys according to some distribution. The amount of different keys, say x, may cover all m keys, or a specific subset of all keys. Formally, this adversarial access pattern can be described as a distribution S:

$$S = (p_1, p_2, \cdots, p_m) \tag{1}$$

where each p_i denotes the fraction of queries for the ith key, $p_1 + p_2 + ... + p_m = 1$. Without loss of generality, we list the keys in monotonically decreasing order of popularity, i.e.,

$$p_1 \geq p_2 \geq \cdots \geq p_m$$

For the case of $x > c$, the adversary can bypass the cache, otherwise no queries would be directed to the back-end nodes. For this case, we can further rewrite the query distribution S as follows:

$$S : \underbrace{p_1 \geq p_2 \geq \cdots \geq p_c}_{cached\ keys} = h \geq \underbrace{p_{c+1} \geq p_{c+2} \geq \cdots \geq p_m}_{uncached\ kyes} \tag{2}$$

Also note that since the c most popular items are served by the front-end cache, to maximize the workload on back-end nodes, the adversary should make the frequency of these keys as low as possible while still keep them the most popular so as to leave as much workload as possible to the back-end nodes. Meanwhile, in these c most popular items, the adversary will not benefit from querying any one item at a higher rate than another. Therefore, the adversary should always query the c most popular items at an equal probability (i.e., $p_1 = p_2 = \cdots = p_c$):

$$S : \underbrace{p_1 = p_2 = \cdots = p_c}_{cached\ keys} = h \geq \underbrace{p_{c+1} \geq p_{c+2} \geq \cdots \geq p_m}_{uncached\ kyes} \tag{3}$$

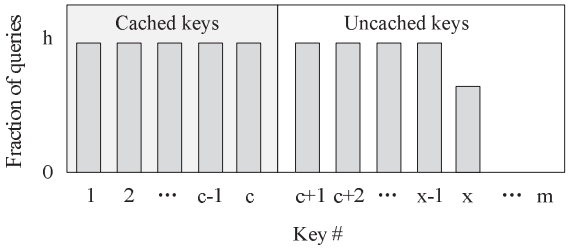


Fig. 2. The best strategy for the adversary to maximize workload on system. The adversary queries x keys, where the rate for $x-1$ keys is the same.

Let L_{max} be the maximum workload on the system (also the maximum load on the back-end nodes). The following theorem states the best strategy for the adversary in term of uncached keys. The theorem was originally proved in [18] and we extend it to the replication scenario. Due to space limitations, we omit the proof here.

Theorem 1: If any distribution S has two uncached keys i and j such that $h \geq p_i \geq p_j \geq 0$, the adversary can always construct a new distribution S' based on S to increase the expectation of the maximum load on the system L_{max}. This new distribution S' is the same as S except $p_i' = p_i + \delta$, $p_j' = p_j - \delta$ where $\delta = min\{h - p_i, p_j\}$.

Theorem 1 intuitively tells us that an adversary can always increase the expectation of the maximum load on system by shifting some load on key j to a more queried key i until this key gets the same fraction as the cached keys. If the adversary applies this process repeatedly, then it will end up with a distribution with the same probability for the first $x-1$ keys (x is the number of keys the adversary queries), or

$$\begin{gathered} S : \underbrace{p_1 = \cdots = p_c}_{cached\ keys} = h = \underbrace{p_{c+1} = \cdots = p_{x-1} \geq p_x}_{uncached\ kyes} > 0 \\ p_{x+1} = \cdots = p_m = 0 \end{gathered} \tag{4}$$

In other words, the best strategy for the adversary to overload the system is to query the first $x-1$ keys (including the cached keys) with the same probability h and the last key with the probability $1-(x-1)h$, as illustrated in Fig. 2.

B. Throughput Bound

Once we know the best strategy for an adversary (also the worst case the system faces), we can calculate the bound of workload on the system. Since keys are randomly partitioned

across back-end nodes, we can model the system as a well-known balls-into-bins model with "the power of two choices", where a ball or item is stored at the least loaded of two (or more) randomly selected bins. In our analysis, keys are treated as balls, and back-end nodes are regarded as bins.

Let us say we are assigning M balls (keys) into N bins (nodes), where each ball is stored at the least loaded of d randomly selected bins. When $M \gg N$, it has been proved in [19] that with high probability the number of balls in any bin is bounded by

$$\frac{M}{N} + \frac{\log\log N}{\log d} \pm \Theta(1) \tag{5}$$

In our model, we have $x - c$ uncached keys that can be considered as balls and n nodes as bins. Setting $M = x - c$ and $N = n$, we have the number of different keys in any node bounded by

$$\frac{x-c}{n} + \frac{\log\log n}{\log d} \pm \Theta(1) \tag{6}$$

Suppose the adversary is sending queries at the rate R, then for each key the query rate is at most $R/(x-1)$, and the maximum load on any node is thus bounded by

$$E[L_{max}] \leq [\frac{x-c}{n} + \frac{\log\log n}{\log d} \pm \Theta(1)] \cdot \frac{R}{x-1} \tag{7}$$

Let $k = \frac{\log\log n}{\log d} \pm \Theta(1)$, we can further rewrite the above bound as follows:

$$\begin{aligned} E[L_{max}] &\leq [\frac{x-c}{n} + \frac{\log\log n}{\log d} \pm \Theta(1)] \cdot \frac{R}{x-1} \\ &= [\frac{x-c}{n} + k] \cdot \frac{R}{x-1} = \frac{R}{n} \cdot \frac{x-c}{x-1} + \frac{kR}{x-1} \\ &= \frac{R}{n} + \frac{R[(1-c)/n + k]}{x-1} = \frac{R}{n} + \frac{R}{n} \cdot \frac{1-c+nk}{x-1} \end{aligned} \tag{8}$$

In the above bound (8), R/n is the best case that the workload is distributed evenly on the back-end nodes, regardless of the load balancing techniques adopted. This value can be used as a baseline to evaluate the attack under a given query rate R and the number of back-end nodes n in the target system. In our work, we also use this value to define the effectiveness of a DDOS attack.

Definition 1: Given a query rate R and the number of back-end nodes n in the target system, the **Attack Gain** of a DDOS is defined as the normalized workload for the most loaded nodes, which is as follows:

$$Attack\ Gain = \frac{E[L_{max}]}{R/n} \tag{9}$$

Definition 2: An **effective** DDOS attack is the one by which an adversary can achieve the attack gain greater than 1.0; Similarly, an ineffecitve DDOS attack is the one with attack gain less than or equal to 1.0.

Returning back to (8), by dividing $E[L_{max}]$ by R/n, we have the normalized workload for the most loaded nodes (also attack gain) being bounded by

$$\frac{E[L_{max}]}{R/n} \leq 1 + \frac{1-c+nk}{x-1} \tag{10}$$

Since $x - 1 > 0$ always holds, it can be seen from (10) that under this most adversarial access pattern, whether the adversary can launch an effective attack depends on the front-end cache size c, the number of back end nodes n, and k. By the definition of k, we can further rewrite it as $k = \frac{\log\log n}{\log d} \pm \Theta(1) = \frac{\log\log n}{\log d} + k'$, where k' denotes a suitable constant.

With regard to the above bound on the normalized workload (attack gain), we have the following two cases:

- **Case 1**: $1 - c + nk > 0$, which means $c < nk + 1 = \frac{n\log\log n}{\log d} + nk' + 1$. Since $x - 1 > 0$ always holds, the adversary should query $x = c + 1$ keys to maximize his gain. Meanwhile, as $\frac{E[L_{max}]}{R/n} > 1.0$, this means that the adversary could always launch an effective attack.
- **Case 2**: $1 - c + nk \leq 0$, which means $c \geq nk + 1 = \frac{n\log\log n}{\log d} + nk' + 1$. In this case, to maximize his gain, the adversary should always query as many keys as possible. That is, he should query $x = m$ keys (the entire key space). Also as $\frac{E[L_{max}]}{R/n} < 1.0$, the adversary can never launch an effective attack.

The above results show that when the cache size is large enough, the best strategy for the adversary is to query all the keys. In other words, under this case, the adversary will never gain advantages by deliberately querying certain amount of keys. However, when the cache size is not so large, the adversary could always query a number of keys that are larger than the cache size so to launch an effective attack.

Our result is quite different from [18], where it is shown that in randomly partitioned cluster services an adversary could always maximize his attack gain by selecting an optimal value of x (the number of queried keys). This optimal value of x is a continuous function of the cache size c and the number of back-end nodes n. Under this optimal value of x, it is shown that the adversary could always launch an effective attack (the normalized maximum workload > 1.0). However, in our analysis we show that given the number of back-end nodes n and the replication factor d, whether the adversary can launch an effective attack depends on the cache size c. With large caches, the adversary can never launch an effective attack.

Our result has a significant meaning for today's cloud service providers. As the required cache size only depends on the number of back-end nodes and the replication factor, and not on the number of items served by the system, it implies that the front-end cache is scalable to the items served. Meanwhile, since $\log\log n / \log d < 2$ holds for almost all deployed clusters (i.e., the number of back-end nodes $n < 10^5$ and the replication factor $d \geq 3$), it implies an $O(n)$ lower bound on the required cache size. Our result indicates that system designers and managers can always protect their clusters from DDOS attacks using a small $O(n)$ fast front-end cache.

Note that in practice, each node can only support limited query rate. If the capacity r_i of each node is larger than

$E[L_{max}]$, then with high probability the adversary will never saturate any node.

IV. Validation

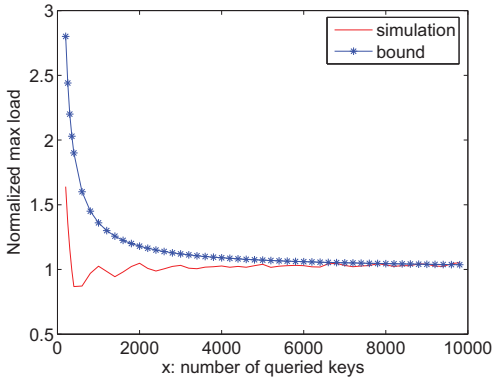
In this section we validate our analysis and results through simulation studies. Our validation will focus on: 1) derive the bound of workload and its tightness; 2) verify the best strategy for the adversary to overload the system, and the required cache size for providing provable DDOS prevention.

To examine the accuracy and tightness of the bound, we simulate a system with 1000 back-end nodes. The replication factor for each item is 3 and the size of queried key space (number of items stored) is 10000. The client launches 1000000 queries per second. For each run of this simulation, x (x>cache size) different keys are queried at the same rate, and the load of the most loaded nodes is recorded. We repeat this simulation for 200 runs, and show the max of the maximum load on the back-end nodes. Fig. 3(a) shows the normalized maximum workload obtained when the cache size is small ($c = 200$), while Fig. 3(b) shows the result when the cache size is large ($c = 2000$). The curve with stars is calculated from Eq. (10) where we set $k' = -1.2$. The figure shows that this bound has a mall gap between numerical results.

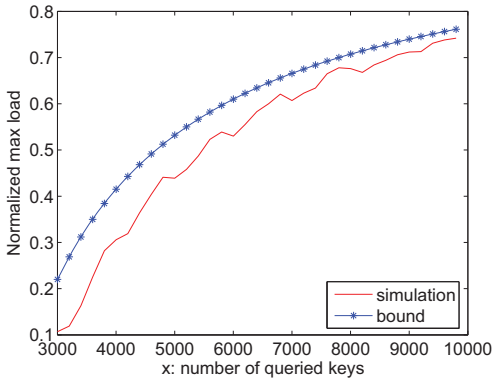
Meanwhile, from Fig. 3, clearly we can see that the workload varies in different ways under different cache sizes. When the cache size is small, i.e., smaller than our bound, the normalized max workload decreases with the increase of the number of queried keys, as that shown in Fig. 3(a). And in this case the adversary could always query a number of keys that are larger than the cache size to launch an effective attack (normalized max load > 1.0); However, from Fig. 3(b) it is observed that when the cache size is large, i.e., larger than our bound, this trend is quite different in that the normalized max workload increases with the increase of the number of queried keys, and the best strategy for the adversary is to query as many keys as possible (all the key space). And it is seen that in this case the adversary can never gain advantages (normalized max load < 1.0). The result shown in Fig. 3 is well in accordance with our analysis.

To verify the best strategy for an adversary, we also compare it with two different workload access patterns: uniform and Zipf. The uniform distribution across all the 10000 keys is expected to be a good case of load balancing and serves as a baseline. The Zipf distribution with parameter 1.01 has a bias towards a few keys and better characterizes the real-word workloads.

Fig. 4 shows the normalized max workload on the back-end nodes under all three different access patterns, where we set the cache size as 100, and vary the number of back-end nodes. From Fig.4 we can see that: 1) the cluster system provides best throughput for the Zipf(1.01) access pattern in which near 80% workloads are concentrated on 20% items. Since these highly concentrated workloads are well served by the front-end cache, the traffic to the back-end nodes are greatly reduced; and 2) when the cache size is large enough as compared to the number of back-end nodes, the cluster system can provide



(a) cache size = 200



(b) cache size = 2000

Fig. 3. Simulation of maximum workload on 1000 back-end nodes with different cache sizes

almost the same throughput under the uniform access pattern and the adversarial access pattern, as at that time all keys are queried at the same rate under both access patterns; however, when the number of back-end nodes becomes large (the front-end cache is relatively small), the workload on system remains stable under uniform access pattern, while it increases under adversarial access pattern. Therefore, we can see that under adversarial access pattern (using our strategy), the adversary can indeed increase the workload on back-end nodes.

Fig. 5(a) shows the best achievable normalized max workload on the system by an adversary under different cache sizes. From Fig. 5(a) we can see that: 1) the normalized max workload that an adversary could achieve decreases with the increase of cache size, since large cache can serve more frequently-queried keys and reduces the workload towards back-end nodes; 2) there is a critical point. When the cache size is smaller than this critical point, the adversary could always launch an effective attack (normalized max load > 1.0); however, when the cache size is larger than this critical point, the adversary can never gain advantages (normalized max load < 1.0); and 3) our bound is tight as it is very close to the critical point.

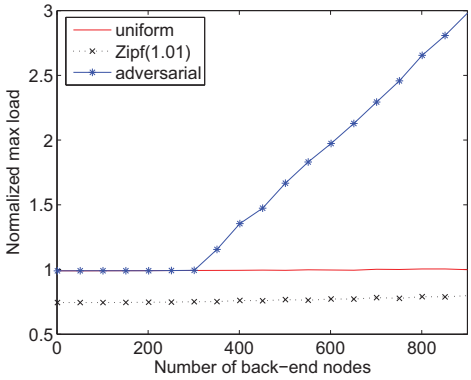
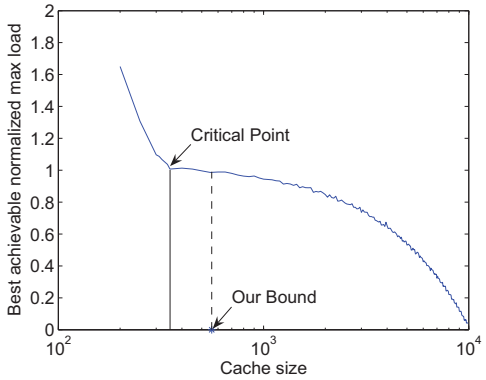


Fig. 4. Normalized max workload on back-end nodes under different access patterns

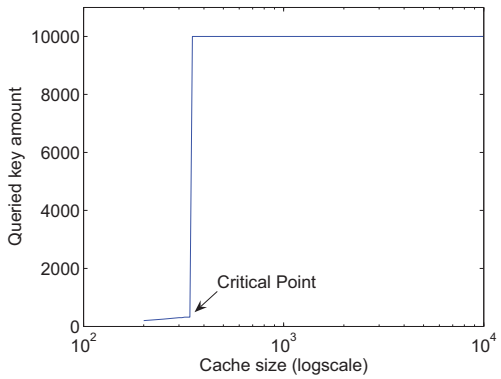
Fig. 5(b) shows the number of keys queried by the adversary under different cache sizes. Since in our simulation, we determine the best normalized max workload by either querying a number of keys that are one more larger than the cache size or querying all keys, the result shown is well in accordance with our result on the best strategy for the adversary.

V. Conclusion

This paper investigates the effectiveness of the front-end cache size on providing provable DDOS prevention for cluster systems with replication. We show that a small and fast popularity-based front-end cache with an $O(n \log \log n / \log d)$ lower bound on the cache size can ensure provable DDOS prevention. For most of today's clusters, our result implies an $O(n)$ lower bound on the required cache size. Our analysis and results are well validated through extensive simulations.



(a) best achievable normalized max workload



(b) number of keys queried by the adversary

Fig. 5. The best max workload and number of keys queried by an adversary under different cache sizes

References

[1] J. Dean and S. Ghemawat, *MapReduce:Simplified data processing on large clusters*, In Proc. 6th USENIX OSDI, Dec. 2004.

[2] Hadoop. http://hadoop.apache.org/, 2011.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat, *A scalable, commodity, data center network architecture*, In Proc. ACM SIGCOMM, Aug. 2008.

[4] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barros, C. Grimes, and S. Quinlan, *Availability in globally distributed storage systems*, In Proc. 9th USENIX OSDI, Oct. 2010.

[5] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, *Finding a needle in Haystack: Facebooks photo storage*, In Proc. 9th USENIX OSDI, Oct. 2010.

[6] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, *Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks*, In Proc. EuroSys, Mar. 2007.

[7] O. O'Malley and A. Murthy, *Winning a 60 Second Dash with a Yellow Elephant*, http://sortbenchmark.org/Yahoo2009.pdf, Apr. 2009.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, *Wide-area cooperative storage with CFS*, In Proc. 18th ACM Symposium on Operating Systems Principles (SOSP), Oct. 2001.

[9] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, *Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web*, In Pro. 29th annual ACM symposium on Theory of computing, 1997.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for Internet applications*, In Proc. ACM SIGCOMM, Aug. 2001.

[11] M. Mitzenmacher, A. W. Richa, and R. Sitaraman. *The power of two random choices: A survey of techniques and results*. In Handbook of Randomized Computing, pages 255-312. Kluwer, 2000.

[12] S. Ghemawat, H. Gobio, and S.-T. Leung, *The Google file system*, In Proc. 19th ACM Symposium on Operating Systems Principles (SOSP), Oct. 2003.

[13] Memcached. *A distributed memory object caching system*. http://memcached.org/, 2011.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, *Dynamo: Amazon's highly available key-value store*, In Proc. 21st ACM Symposium on Operating Systems Principles (SOSP), Oct. 2007.

[15] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, *FAWN: A fast array of wimpy nodes*, In Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), Oct. 2009.

[16] HBase. http://hbase.apache.org/, 2011.

[17] F. Chang, J. Dean, S. Ghemawat, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, *Bigtable: A distributed storage system for structured data*, In Proc. 7th USENIX OSDI, Nov.2006.

[18] B. Fan, H. Lim, D.G. Andersen, M. Kaminsky, *Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services*, In Proc. 2011 ACM Symposium on Cloud Computing (SOCC'11), Oct. 2011, Cascais, Portugal.

[19] P. Berenbrink, A. Czumaj, A Steger, and B. Vöcking, *Balanced Allocations: The Heavily Loaded Case*, In Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC'00), 2000.