Load balancing for minimizing the average response time of get operations in distributed key-value stores

Antonios Makris, Konstantinos Tserpes, Dimosthenis Anagnostopoulos
Dept. of Informatics and Telematics
Harokopio University of Athens
Tavros, Greece
{amakris,tserpes,dimosthe}@hua.gr

Jörn Altmann
Technology Management, Economics, and Policy
College of Engineering
Seoul National University
Seoul, South Korea
jorn.altmann@acm.org

***Abstract*—We investigate the impact of an unevenly distributed load among nodes of a distributed key-value store on response times. We find that response times of "get" operations quickly degrade in the presence of power law distributions of load and identify the point, at which the system needs to apply a mitigation approach. The migration technique, which we propose, overcomes the long response times of consistent hashing placement techniques. Our technique is a hybrid approach that combines consistent hashing and a directory for exceptions. Our experimental results show an improvement in the average response times and an equal load among the nodes.**

Keywords—resource allocation; hash tables; directory look-ups; load balancing; Zipf distribution of get operations; responsed time optimization; simulation

I. INTRODUCTION

Distributed hash tables (DHT) are a standard approach for the placement of data objects among the nodes of a distributed data store. A DHT is a class of a decentralized distributed system that provides a lookup service similar to a hash table. Data are stored and managed in the form of key-value pairs, and every node in the system can efficiently retrieve the object associated with a given key through the DHT. The placement of data objects is achieved through specialized partition algorithms that assign the responsibility to nodes to manage each object. As node-to-node systems can be highly heterogeneous, the distribution of objects through the selection of random identifiers for data objects can lead to imbalances. Besides, these systems do not take into consideration uneven request workloads (e.g., existence of nodes that hold popular objects) and, therefore, can lead to heavy loads on those nodes. As overloaded nodes become hotspots in the system, the overall system performance may significantly decrease.

To address the problem of load balancing several approaches have been proposed. Many of those suggest the distribution of the objects among nodes in such way that every node in the system will maintain the same amount of keys. Load-stealing, load-shedding, object migration and bin packing are some of these methods. A typical characteristic of these methods is that they consider only the number of objects that each node holds. They do not deal with the dynamics of the system, or the variations in the popularity of objects. Empirical studies have shown that requests in a P2P system follow a Zipf distribution [1]. This implies that there is a small number of specific data objects that are requested with a much higher frequency than the vast majority of the objects stored in the system.

In this study, we demonstrate the impact of an uneven distribution of the frequency of data requests ("get" operations) across system nodes on the system performance. Our experiments show a deterioration of the response times in nodes serving a high rate of get operations, leading to contract term violations.

Furthermore, in order to deal with this challenge, our solution employs data object migration technique when a decreased performance is detected. In particular, our solution represents a new load balancing approach, which combines consistent hashing and a directory for exceptions. It has been implemented in a Redis cluster (with an in-memory key-value store [2]) under data object request workloads exhibiting a power law distribution. Our approach measures the load of each node in terms of the number of requests and, when the load exceeds a certain threshold, the

978-1-5090-4429-0/17/$31.00 ©2017 IEEE.

migration procedure begins. The threshold itself is set on the basis of the "permissible" limit of response time in the Redis infrastructure Requests and corresponding keys are migrated from overloaded nodes to less loaded ones, such that the system load gets more evenly distributed and the system average response time decreases. For selecting a less loaded node, a variation of a bin-packing algorithm, the First Fit Decreasing (FFD) algorithm, has been chosen.

To measure the performance of our approach, we conducted experiments on a Redis cluster infrastructure. Experiments show a significant reduction in the average response times for get operations in the system and an evenly distributed load in the nodes. The average load in an overloaded node decreases by 60%, and it is transferred to the nodes that still operate under normal conditions. Finally, the response time in the overloaded node is decreased by 33% while the shift of load to the rest of the nodes led to an increase of a 1.3 factor.

The contribution of this work is to highlight an inherent weakness of consistent hashing, which negatively impacts the performance of distributed key value stores. Moreover, the evaluation results of our solution (i.e., the combination of consistent hashing with directory-based indexing) suggest our solution to be an efficient migration approach.

The document is structured as follows: Section II provides details about the related work in load balancing in peer-to-peer systems and distributed hash tables; Section III describes the problem formulation and the load balancing algorithm; Section IV describes the system architecture; Section V gives a full account of the experimental results while Section VI presents the final conclusions of this study and future work.

II. RELATED WORK

The most widespread method for load balancing is consistent hashing (CH), which was initially presented in [3] and employed by the majority of the NoSQL data stores. CH distributes data and incoming requests to the nodes equally. Although this technique solves the problem of data partition, it also comes with three drawbacks. First, CH destroys locality, which is an important factor for supporting range queries. Second, in case of events, in which high loads occur due to client requests around a specific area, CH may cause the system infrastructure to fail serving and responding to a high volume of incoming requests. Third, client requests may fall into a specific node, thus leading to hotspots and increasing the response times. CH extensions, which try to address these shortcomings, are presented in the following paragraphs.

Load balancing in a distributed hash table (DTH) can be accomplished either with the use of a hash function that randomize the address of the table either by making each participant node responsible for a balanced portion of the table address space. However, some nodes may be responsible for a larger portion of the addresses and consequently for a larger number of objects, and also in cases that application has to support range queries, the distribution of objects in the address space cannot be randomized. In order to overcome these challenges two balancing algorithms are presented [3]: a) address-space balancing that balances the distribution of the key address space to nodes with the use of virtual nodes and b) item balancing that balance the distribution of objects among nodes.

As systems can be highly heterogeneous, a random distribution of objects among peers can be lead to imbalance. In order to address these imbalances, different schemes have been proposed [4] [14] [15]. Some load balancing schemes transfer virtual servers from heavily loaded nodes to lightly loaded nodes.

In the standard hash table implementation, the hash function map keys to certain entries of a table. If more than one keys map to the same entry of that table, then these keys are stored in a linked listed (chain). Thus, the time to fetch a key is proportional to the length of the longest chain in table. To cope with this problem, the two-choice paradigm that conducts the standard balls and bins problem is represented in [6]. The most common methods based in two-choice paradigm are: a) load-stealing where a less loaded peer seeks out load to take from more heavily loaded peers and b) load-shedding where an overloaded peer attempts to offload work to a less loaded peer [7]. Two random hash functions are used, to pick candidate bins for each object to be inserted. As a result, the time to search the hash table is significantly reduced [6].

Most placement algorithms in P2P systems do not take into consideration uneven requests to particular nodes (hotspots) that can lead to heavy lookup traffic load. In [8], two methods are proposed for load balancing, by taking into account object popularity and routing. These methods are: routing table reorganization where overloaded nodes are moved from other nodes routing tables and caching where objects are replicated and cached on another node in order to reduce its request load. The impact of this technique is that the number of received requests at the nodes holding popular objects is minimized, by sharing the request traffic for each cached object among the node owning the object and the nodes holding the replicas.

III. Problem Formulation

A. *Load balancing algorithm*

A hash function like CRC16 can distribute evenly the keys that are generated based on a uniform distribution (i.e., each node receives an equal share of initial put operations). Even though this seems to be the case for the operations that persist new data, there is no indication that this is the case for load imposed through get operations, if the number of get operations increases beyond a threshold [1].

At this point, an exponential increase of the response times occurs. It follows a power law distribution. This happens even though current distributed store implementations demonstrate a remarkable ability to preserve performance under very heavy loads.

Henceforth, the term "node load" or "load" will be used to denote the number of client requests within a time interval, whereas the term "request" refers to a 'get' operation. As such, every node has a capacity, which corresponds to the maximum load that it can serve without reaching a threshold defined in terms of average response time.

As depicted in Figure 1, the average response time in an experimental Redis cluster shows an exponential increase in case of request rates greater than 20000 get operations per second.

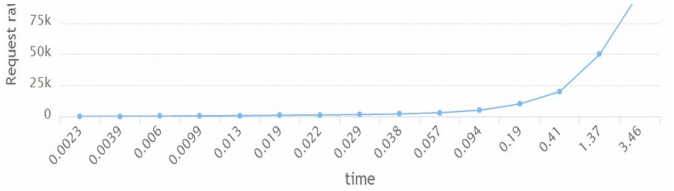


Fig. 1. Response time in seconds to get one particular key, as the request rate increases in a Redis cluster.

This declination in the performance is not affected by the amount of keys kept in a node nor the data object characteristics as it was shown experimentally in [9]. It takes extreme conditions for these to contribute to the performance degradation, and, even then, there are mitigation plans available(e.g., use of SCAN with complexity O(1) [10]). With respect to get the operations' rate, this is not the case [11]. Twitter, which uses Redis, served 300 million users during the year 2015, which implies 300 million keys for the user profiles alone.

Considering this situation, we propose a solution that identifies the point, at which the performance degrades and, then, triggers a mitigation policy, avoiding an exponential increase of the response times. The objective of our solution is to minimize the average response time of the system under a continuously changing load due to get operations. In other words, the research problem can be defined as:

Given that each node has a limited capacity of managing rates of get requests, which keys should be selected for migration and to which node should they be migrated in the presence of uneven load distributions?

This problem can be deduced to a variation of the bin-packing problem, in which data objects need to be distributed equally across all bins (nodes).

If the load exceeds a threshold (T) at any node, then objects are migrated from this overloaded node to less loaded ones, in order to reduce the response time and balance the load in all nodes. The selection of keys to be migrated and the new nodes, which will host the keys, are based on the First Fit Decreasing algorithm (FFD), that addresses the bin-packing problem [12]. To make this feasible, the frequencies of get operations are monitored and are kept in a list in decreasing order. With the use of the

FFD algorithm, each object is stored into the first bin (node), which fits, based on its remaining capacity. Following this, the steps of our algorithm can be described as:

```
2:   while true do
3:     for each i ∈ M do
4:       Compute RT_i and NR_i
5:     end for
6:     for each i ∈ M do
7:       if max(NR_i) >T then
8:         Find (Node_i, min(NR_i))
9:         Find (key, max(FrRk_i(key)))
10:        Migrate (key, Node_i)
11:        addException (key, Node_i)
12:      end if
13:    end for
14:  end while
15: end procedure
```

where RT_i and NR_i are the average response time and the number of request for node #i, respectively, and $FrRk_i(key)$ is the frequency of requests for a specific key in node #i.

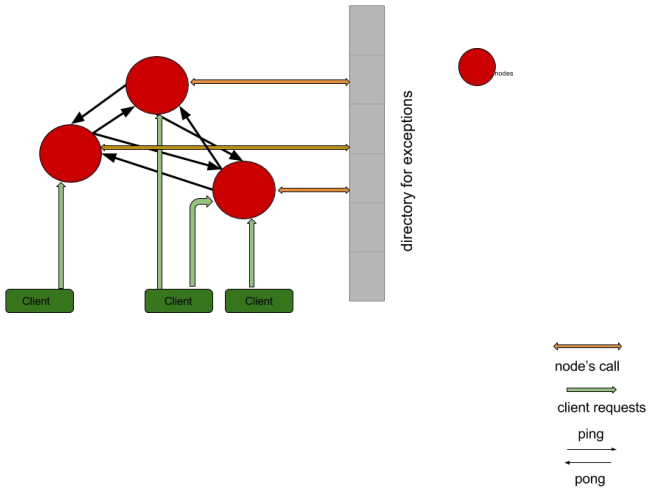
IV. System Architecture

Key-value stores, such as Redis, expect a client to mediate between the cluster and the system component that performs the operations (e.g., a frontend). In the Redis terminology, these clients are called "superclients" as they are aware of the hash function that each nodes implements. This means that – despite their names – they are expected to be a part of the server infrastructure and not part of the client. Without the existence of such a superclient, the actual clients would have to randomly select a node to perform an operation and the node should decide (through the application of the hash function to the key in question) whether the operation should be addressed by itself or another node. This is done through a gossip protocol that is used to exchange information between nodes, to send ping packets and to propagate cluster messages needed to signal specific conditions [13].

The proposed implementation needs to support a migration process which is not possible by default through consistent hashing techniques. Instead a directory-based approach would be more meaningful, so the keys can be mapped to nodes without any particular restriction. However directory-based solutions are not preferred, mainly because of the overhead to efficiently manage potentially huge directories, the long lookup times and the need of centralized approaches. To this end, we employ a hybrid approach, in which consistent hashing is the method of data placement but a shared decentralized directory is maintained for exceptions.

In a Redis cluster architecture, cluster nodes are not able to proxy requests, thus the clients need to send requests, redirects, and communication messages to other nodes through redirection errors (i.e., MOVED and ASK) based on ASCII protocols [2]. Thus, the client may be ignorant of the cluster state and employ the redirection property to reach any key. To achieve a better performance, we created a shared decentralized directory for exceptions called "smart client" (Figure 2). If a request arrives in a random node in the cluster and this node holds the key, the node serves the client directly. Otherwise, it raises an exception to the directory and redirects the request to the correct node. This way, the nodes are able to cache the map between keys and nodes and with a process similar to machine learning the cluster nodes are able to answer the requests without MOVED redirection errors and to acquire knowledge of what objects they hold. Due to this, a better overall performance and average response time can be expected.

Fig. 2. Cluster architecture.



V. EXPERIMENTS

B. *Experiment set-up*

The underlying computing infrastructure that has been used for the simulation has been a commodity machine with the following configuration: Ubuntu 14.04 LTS 64-bit; Intel® CoreTM i5-4570 CPU @ 3.20GHz × 4; and 7.7 GiB RAM.

For all experiments, the following assumptions have been made:

- Five consecutive separate calls are conducted, in order to gather the experimental results.
- Twenty keys are requested in each call based on a Zipf distribution
- The request rate is 1000 calls per key, following a power law distribution.

The cluster setup configuration consists of 3 nodes and is set up in the abovementioned machine using virtualized nodes. Although Redis maintains a slave node for each master for fault tolerance purposes, it does not make any difference for our experiments.

The first experiment demonstrates that in a set up that includes a) a uniformly distributed number of keys across all three nodes, and; b) a system-level request rate that obeys Zipf's law, the load is also unevenly distributed within nodes. In particular, node (M1) receives many more requests than M2 and M3 as it keeps more popular keys. Figure 3 illustrates the average load of the master nodes.

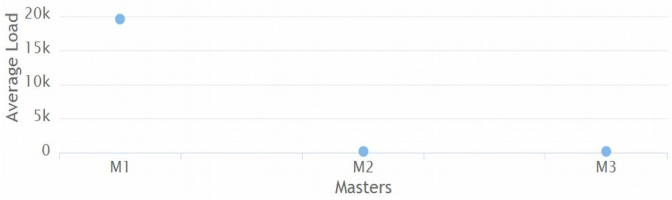


Fig. 3. Average Load of master nodes.

Based on the selected settings in the power law distribution, for the aggregated five calls, the 98% of the keys are requested from M1 as it is shown in Figure 4.

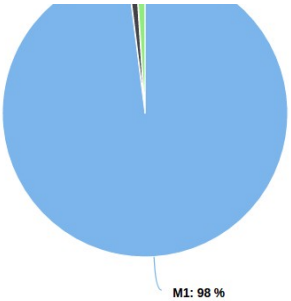


Fig. 4. Number of keys requested from master nodes.

This uneven distribution of the load results in degraded average response times. Figure 5 demonstrates the respective average response times for each of the nodes.

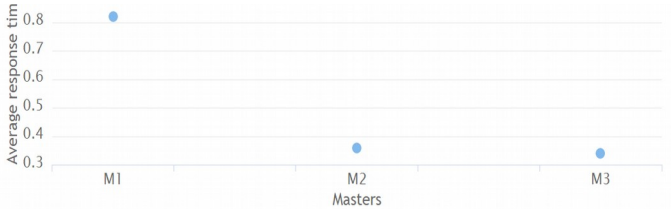


Fig. 5. Average response time of the three master nodes in seconds for fetching the requested keys.

We observe that the average response time varies significantly between the nodes in the cluster. The majority of the overall system load is served by one node and the other nodes in the cluster are almost idle. As a result, the response time increased dramatically, leading to poor performance and potential breaches of service level agreement clauses.

Consequently, we can state that there is the need of a load balancing technique that is able to distribute the requests among the involved nodes in the system. The algorithm should ensure a uniform distribution of the load in the cluster by migrating keys and requests from overloaded nodes to nodes with a normal load.

C. *Load balancing solution*

In the next experiments, we apply our load balancing solution that takes into account the request popularity. Initially, using the FFD algorithm, selected keys are moved to nodes operating beyond the defined threshold. Figure 6 illustrates the keys' distribution after the application of FFD.

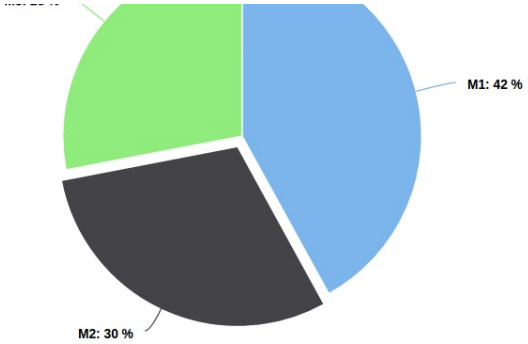


Fig. 6. Number of keys requested from master nodes.

Figure 6 demonstrates that keys are almost equally distributed among the system nodes. Popular objects are migrated from the overloaded node M1 to the other less loaded nodes. In order to bypass the hash function that will continue to point to M1 (even in the case of migrated components), we extend the operation of the smart client by keeping a directory that contains only the exceptions. Due to the low number of exceptions, the directory is small in length, achieving low lookup times.

Figure 7 demonstrates the average load in each master node after applying the load balancing algorithm and with no load balancing.

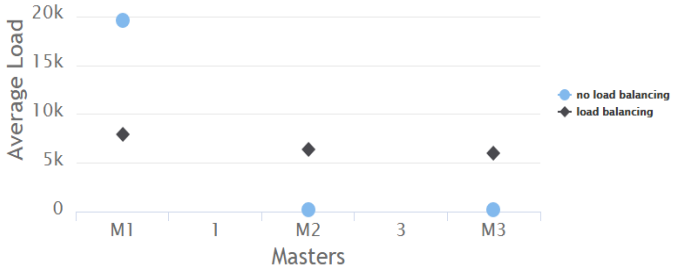


Fig. 7. Average load of master nodes with load balancing and with no load balancing.

According to Figure 7, the average load in each node presents smaller fluctuations than the baseline (blue dots, also see Figure 3).The rate of requests is evenly distributed in all nodes (black diamonds), applying a fair-share approach and thus minimizing the probability to have overloaded nodes. Therefore, the response times are expected to be largely improved in M1 and worsen in M2 and M3. The extent to which the latter happens is crucial for the success of the endeavor. Figure 8 illustrates the average response times for each node after applying the load balancing algorithm and with no load balancing.

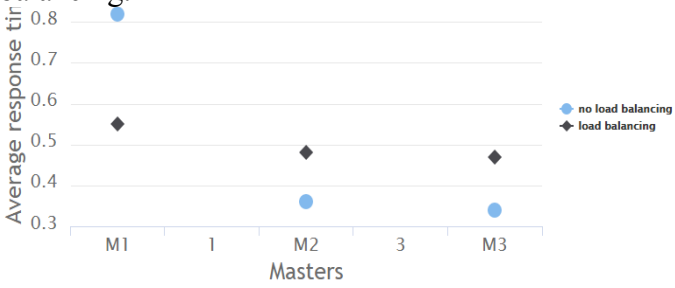


Fig. 8. Average response time of system nodes with load balancing and with no load balancing.

As we can see, the response time in the overloaded node M1 presents a conspicuous reduction while the response time of the other nodes increased only by a small extent. Of course this behavior was expected, as the load was transferred from M1 to the lessloaded nodes M2, M3.

In order to absolute differences between the two approaches, Table 1 presents a comparison of the average load, average response time, and keys' distribution with load balancing mechanism and without load balancing (baseline).

TABLE I. AVERAGE LOADS, KEYS DISTRIBUTIONS, AND AVERAGE RESPONSE TIME.

	No Load Balancing			Load Balancing		
	M1	**M2**	**M3**	**M1**	**M2**	**M3**
Average Load	19600	200	200	7900	6400	6000
Keys Distribution	98%	1%	1%	42%	30%	28%
Average Response Time	0,82	0,36	0,34	0,55	0,48	0,47

The experimental results show that with our load balancing solution a uniform distribution of load is achieved. The average load on the overloaded node M1 decreased by 60% and has been transferred to the nodes operating under normal load. The keys that were transferred are almost equally distributed across the two nodes M2 and M3. Although the load in the nodes M2 and M3 increased about 32 times in order to serve the new arrival keys and relieve M1, it got compensated by a significant reduction in the response times. We observe that the response time in M1 is decreased by 33% and, in the other two

nodes, the response time increased only about 1.3 times.

With the proposed load balancing solution, we achieved to balance the average load between the nodes in the system. Keys are transferred to less loaded nodes and thus a homogeneous distribution in the system load could be achieved. Furthermore, clients are served from nodes with low load and received responses for get queries faster.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented an analysis on the load in a node-to-node system and specifically in Redis. We conducted a number of experiments in a Redis deployment under a certain setup and examined how the frequency of queries affects response times and how an equal load distribution can lead to a performance increase.

As our analysis showed, uneven request workloads to nodes that hold popular objects can lead to heavy load. Based on this result, we proposed a new approach for balancing the system load by considering the workload caused by client requests. Requests and corresponding objects (keys) are migrated from overloaded nodes to less-loaded ones. The results show that, with our algorithm, the average response time decreases significantly. The average response time is also uniform across the nodes. Thus, the performance is increased.

Our future plan is to deploy a Redis cluster in a cloud infrastructure, in order to explore the expected behavior of our algorithm in a large scale, making the experiments more realistic.

ACKNOWLEDGMENT

This work is partly funded by the research project "Middleware development for the optimal replica placement in distributed key-value stores". This research was also partly conducted within the project BASMATI, which has received funding from the ICT R&D program of the Korean MSIP/IITP (R0115-16-0001) and from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 723131.

REFERENCES

[1] A. Gupta, P. Dinda, and F. Bustamante, "Distributed popularity indices," in *Proc. of ACM SIGCOMM*, pp. 2–3, 2005.

[2] Redis, "Redis cluster specification – Redis," *Redis Documentation*, 2015. [Online]. Available: https://redis.io/topics/cluster-spec.

[3] D. R. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," *Proc. Sixt. Annu. ACM Symp. Parallelism algorithms Archit. SPAA 04*, vol. 39, no. 6, pp. 36, 2004.

[4] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I, "Load balancing in structured P2P systems," *Peer-to-Peer Syst.*, vol. 1, no. 4, pp. 100–103, 2003.

[5] J. Aspnes, J. Kirsch, and A. Krishnamurthy, "Load balancing and locality in range-queriable data structures," in *Proc. of the 23ed Annual ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pp. 115, 2004.

[6] M. Mitzenmacher, A. W. Richa, R. Sitaraman, "The power of two random choices: A survey of techniques and results," in: S. Rajasekaran et al.: *Handbook of Randomized Computing*, vol. 1, pp. 255–312, 2001.

[7] J. Byers, J. Considine, and M. Mitzenmacher, "Simple Load Balancing for Distributed Hash Tables," *Peer-to-Peer Syst. II*, pp. 80–88, 2003.

[8] S. Bianchi, S. Serbu, P. Felber, and P. Kropf, "Adaptive Load Balancing for," *Statistics (Ber)*, pp. 411-418, 2009.

[9] A. Makris, K. Tserpes, and D. Anagnostopoulos, "Load Balancing in In-Memory Key-Value Stores for Response Time Minimization."

[10] Redis, "SCAN – Redis," 2015. [Online]. Available: https://redis.io/commands/scan.

[11] www.statista.com, "Twitter: number of active users 2010-2016 | Statista." 2016.

[12] G. Dósa and J. Sgall, "First Fit bin packing: A tight analysis," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 20, 2013.

[13] Redis, "Redis cluster tutorial – Redis," 2015. [Online]. Available: https://redis.io/topics/cluster-tutorial.

[14] D. M. Quan and J. Altmann, "Resource allocation algorithm for the light communication grid-based workflows within an SLA context," Intl J of Parallel, Emergent and Distributed Systems, vol.24, no.1, pp. 31-48, 2009.

[15] D. Kyriazis, K. Tserpes, A. Menychtas, A. Litke, and T. Varvarigou, "An innovative workflow mapping mechanism for grids in the frame of quality of service," Future Generation Computer Systems 24 (6), pp. 498-511, 2008.