R. Yanggratoke, G. Kreitz, M. Goldmann, R. Stadler, and V. Fodor. "On the performance of the Spotify backend." *Journal of Network and Systems Management*, 2013.

On the performance of the Spotify backend

Rerngvit Yanggratoke · Gunnar Kreitz · Mikael Goldmann · Rolf Stadler · Viktoria Fodor

October 7, 2013

Received: 16 March 2013 / Revised: 25 September 2013 / Accepted: 2 October 2013

Abstract We model and evaluate the performance of a distributed key-value storage system that is part of the Spotify backend. Spotify is an on-demand music streaming service, offering low-latency access to a library of over 20 million tracks and serving over 20 million users currently. We first present a simplified model of the Spotify storage architecture, in order to make its analysis feasible. We then introduce an analytical model for the distribution of the response time, a key metric in the Spotify service. We parameterize and validate the model using measurements from two different testbed configurations and from the operational Spotify infrastructure. We find that the model is accurate—measurements are within 11% of predictions—within the range of normal load patterns. In addition, we model the capacity of the Spotify storage system under different object allocation policies and find that measurements on our testbed are within 9% of the model predictions. The model helps us justify the object allocation policy adopted for Spotify storage system.

Keywords Key-value store · distributed object store · object allocation policy · performance modeling · performance measurements · response times

Rerngvit Yanggratoke (✉) · Rolf Stadler · Viktoria Fodor
ACCESS Linnaeus Center, KTH Royal Institute of Technology
Osquldas väg 10, SE-100 44, Sweden
E-mail: {rerngvit,stadler,vjfodor}@kth.se

Gunnar Kreitz · Mikael Goldmann
Spotify AB, Birger Jarlsgatan 61, SE-113 56, Sweden
E-mail: {gkreitz,migo}@spotify.com

1 Introduction

Spotify is an on-demand music streaming service (www.spotify.com), offering low-latency access to a library of over 20 million tracks and serving over 20 million users currently [1]. The core function of the Spotify service is audio streaming, which is provided by the Spotify storage system, with backend servers located at three sites (Stockholm, Sweden, London, UK, and Ashburn, VA). The Spotify backend servers run even a number of related services, such as music search, playlist management, and social functions. The Spotify service is a peer-assisted system, meaning it has a peer-to-peer component to offload backend servers. When a client plays a music track, its data is obtained from a combination of three sources: the client local cache (if the same track has been played recently), other Spotify clients through peer-to-peer technology, or the Spotify storage system in a backend site [1].

Low latency is key to the Spotify service. When a user presses "play", the selected track should start "instantly." To achieve this, the client generally fetches the first part of a track from the backend and starts playing as soon as it has sufficient data so that buffer underrun ("stutter") will be unlikely to occur. Therefore, the main metric of the Spotify storage system is the fraction of requests that can be served with latency at most t for some small value of t, typically around 50 ms. (We sometime use the term response time instead of latency in this paper.) The Spotify storage system has the functionality of a (distributed) key-value store. It serves a stream of requests from clients, whereby a request provides a key and the system returns an object (e.g., a part of an audio track).

In this paper, we present an analytical model of the Spotify storage architecture. The model allows us to estimate the distribution of the response time of the storage system as a function of the load, the storage system configuration, and model parameters that we measure on storage servers. The model centers around a simple queuing system that captures the critical system resource (i.e., the bottleneck), namely, access to the server's memory cache and disk where the objects are stored. We validate the model (1) for two different storage system configurations on our laboratory testbed, which we load using anonymized Spotify traces, and (2) for the operational Spotify storage system, whereby we utilize load and latency metrics from storage servers of the Stockholm site. We find that the model predictions are within 11% of the measurements, for all system configurations and load patterns within the confidence range of the model. As a consequence, we can predict how the response time distribution would change in the Stockholm site, if the number of available servers would change, or how the site in a different configuration would handle a given load. Overall, we find that a surprisingly simple model can capture the performance of a system of some complexity. We explain this result with two facts: (1) the storage systems we model are dimensioned with the goal that access to memory/storage is the only potential bottleneck, while CPUs and the network are lightly loaded; (2) we restrict the applicability of the model to systems with small queues —they contain at most one element on average. In other words, our model is accurate for a lightly loaded system.

The model captures well the normal operating range of the Spotify storage system, which has strict response time requirements. In addition, the system is dimensioned to absorb sudden load surges, which can be caused by, e.g., one of the Spotify backend sites failing, or the service being launched in the new region.

Further, we model a capacity of a storage cluster under two different object allocation policies, the popularity-aware policy, where objects are allocated to servers according to their popularity, with the aim of load balancing, and the random policy, where objects are allocated uniformly at random across servers. We validate the model on our laboratory testbed and find that the model predictions are within 9% of the measurements, for the parameter range investigated. The model suggests that the capacity of the Spotify storage system under both policies will be (approximately) the same. This justifies the Spotify design, which adopts the random policy, which is easier to implement in the key-value storage system than the popularity-aware policy.

This work is a significant extension of a conference publication presented at CNSM 2012 [2]. It includes now an analytical model for estimating the capacity of a storage cluster for different object allocation policies, together with its validation on our testbed. In addition, the related work section is extended, and numerous small improvements have been made and clarifications have been added to the text in [2].

The paper is organized as follows. Section 2 describes the Spotify storage system. Section 3 contains the analytical model for predicting distribution of response time and its validation, both on the lab testbed and on the Spotify operational system. Section 4 describes the model for estimating the capacity of a storage cluster under two different allocation policies and contains the validation results. Section 5 discusses related work and Section 6 contains our conclusions and future work.

2 The Spotify backend storage architecture

We give a (simplified) overview of the part of the Spotify backend responsible for music delivery to the clients. Its architecture is captured in Fig. 1. Each Spotify backend site has the same layout of storage servers, but the number of servers varies. The master storage component is shared between the sites. When a user logs in, the client connects to an *Access Point* (AP) using a proprietary protocol. Through the AP, the client can access the backend services including storage. The client maintains a long-lived TCP connection to the AP, and requests to backend services are multiplexed over this connection.

Spotify's storage is two-tiered. A client request for an object goes to *Production Storage*, a collection of servers that can serve most requests. The protocol between the AP and Production Storage is HTTP, and in fact, the Production Storage servers run software based on the caching Nginx HTTP proxy [3]. The objects are distributed over the production service machines using consistent hashing of their respective keys [4]. Each object is replicated on three different servers, one of which is identified as the primary server for the object. APs

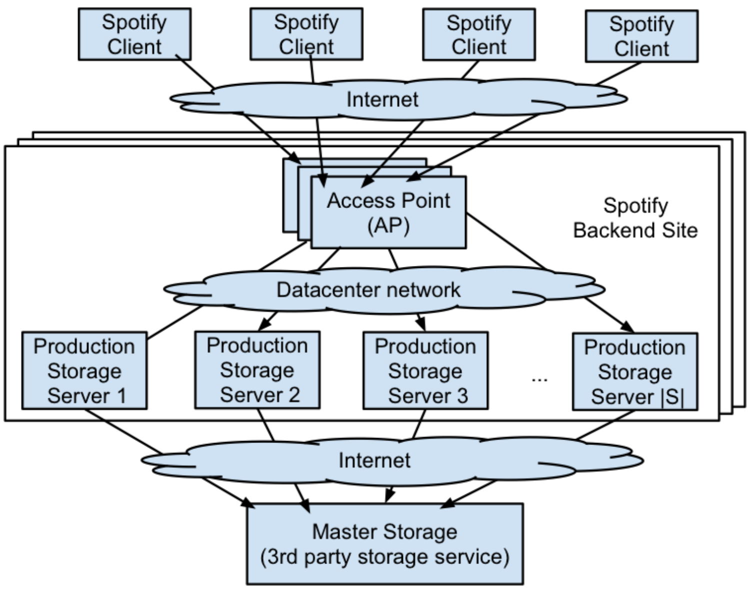

Spotify Client
Spotify Client
Spotify Client
Spotify Client
Internet
Access Point (AP)
Spotify Backend Site
Datacenter network
Production Storage Server 1
Production Storage Server 2
Production Storage Server 3
...
Production Storage Server |S|
Internet
Master Storage (3rd party storage service)

Fig. 1 Spotify Storage Architecture

route a request for an object to its primary server. If the primary server fails or does not store the requested object, the server will request it from one of the replicas. If they do not store it, the request will be forwarded over the Internet to *Master Storage* (which is based upon a third-party storage service) and the retrieved object will subsequently be cached in Production Storage.

A key performance metric for Spotify is playback latency, and the system design is intended to provide low latency for the vast majority of requests coming from a large, and growing, user base. By using a proprietary protocol with long-lived connections, the latency overhead of a connection establishment (DNS, TCP handshake, TCP windows) is avoided. The design with a two-tiered storage arises from the engineering problem of providing low-latency access for most requests to a growing catalogue of music, with a popularity distribution close to Pareto. The Production Storage tier gives low-latency access to almost all requests, using high performance servers, while the Master Storage tier provides a large storage capacity for all information at a higher latency.

While the presentation here centers on music delivery, we remark that the storage system delivers additional data to Spotify clients, in particular images (e.g., cover art for albums) and advertisements. We also point out that the number of requests that a client makes to the backend storage for each track played varies significantly, due to local caching and peer-to-peer functionality. As a consequence, the request rates presented in this paper do not correspond to the actual number of tracks played in the Spotify system.

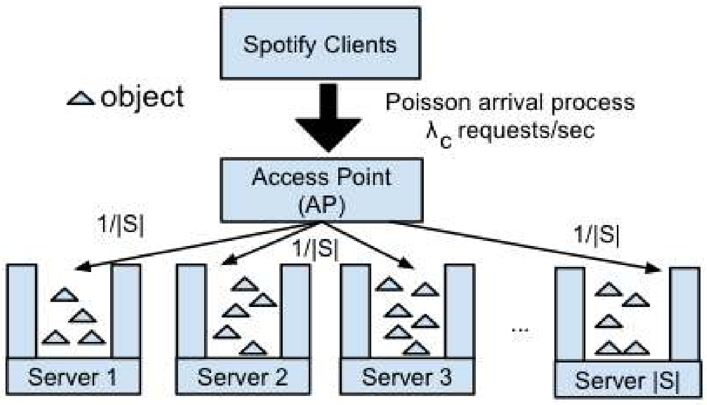


Fig. 2 Simplified architecture as a basis for the performance model

3 Predicting response time

In order to make a performance analysis tractable, we develop a simplified model of the Spotify storage architecture (Fig. 1) for a single Spotify backend site, the result of which is shown in Fig. 2. First, we omit Master Storage in the simplified model and thus assume that all objects are stored in Production Storage servers, since more than 99% of the requests to the Spotify storage system are served from the Production Storage servers. Second, we model the functionality of all APs of a site as a single component. We assume that the AP selects a storage server uniformly at random to forward an incoming request, which approximates the statistical behavior of the system under the Spotify object allocation and routing policies. Further, we neglect network delays between the AP and storage servers and the queuing delays at the AP, because they are small compared to the response times at the storage servers. In the following, we analyze the performance of the model in Fig. 2 under steady-state conditions, Poisson arrivals of requests, and exponentially distributed service times.

3.1 Modeling a single storage server

In the context of a storage system, the critical resources of a storage server are memory and disk access, which is captured in Fig. 3. When a request arrives at a server, it is served from memory with probability q and from one of the disks with probability $1 - q$. Assuming that the server has n_d identical disks, the request is served from a specific disk with probability $1/n_d$. We further assume that requests arrive at the server following a Poisson process with rate λ. (All rates in this paper are measured in requests/sec.) We model access to memory or a disk as an $M/M/1$ queue. We denote the service rate of the memory by μ_m and that of the disk by μ_d, whereby $\mu_m \gg \mu_d$ holds.

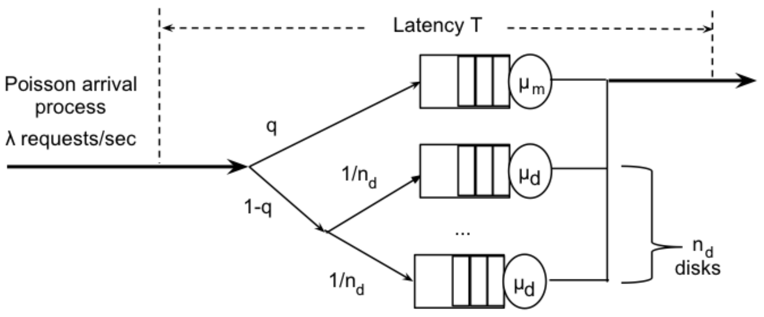


Fig. 3 Queuing model of critical resources on a server

Based on Fig. 3, we compute the latency T for a request on a server, which we also refer to as response time. (In queuing systems, the term *sojourn time* is generally used for T.) The probability that a request is served below a specific time t is given by $Pr(T \leq t) = qPr(T_m \leq t) + (1-q)Pr(T_d \leq t)$, whereby T_m and T_d are random variables representing the latency of the request being served from memory or a disk, respectively. For an $M/M/1$ queue in steady state with arrival rate λ, service rate μ, and latency T, the formula $Pr(T \leq t) = 1 - e^{-\mu(1-\lambda/\mu)t}$ holds [5]. Therefore, we can write

$$Pr(T \leq t) = q(1 - e^{-\mu_m(1-\lambda_m/\mu_m)t}) + (1-q)(1 - e^{-\mu_d(1-\lambda_d/\mu_d)t}),$$

whereby λ_m is the arrival rate to the memory queue and λ_d to a disk queue. Typical values in our experiments are $t \geq 10^{-3}$ sec, $\lambda_m \leq 10^3$ requests/sec, and $\mu_m \geq 10^5$ requests/sec. We therefore approximate $e^{-\mu_m(1-\lambda_m/\mu_m)t}$ with 0. Further, since $\lambda_m = q\lambda$ and $\lambda_d = (1-q)\lambda/n_d$ hold, the probability that a request is served under a particular latency t is given by

$$Pr(T \leq t) = q + (1-q)(1 - e^{-\mu_d(1-(1-q)\lambda/\mu_d n_d)t}). \tag{1}$$

3.2 Modeling a storage cluster

We model a *storage cluster* as an AP and a set S of storage servers, as shown in Fig. 2. The load to the cluster is modeled as a Poisson process with rate λ_c. When a request arrives at the cluster, it is forwarded uniformly at random to one of the storage servers. Let T_c be a random variable representing the latency of a request for the cluster. We get that $Pr(T_c \leq t) = \sum_{s \in S} \frac{1}{|S|} Pr(T_s \leq t)$, whereby T_s is a random variable representing the latency of a request for a storage server $s \in S$.

For a particular server $s \in S$, let $\mu_{d,s}$ be the service rate of a disk, $n_{d,s}$ the number of identical disks, and q_s the probability that the request is served from memory. Let $f(t, n_d, \mu_d, \lambda, q) := Pr(T \leq t)$ defined in Equation 1. Then, the probability that a request to the cluster is served under a particular latency t is given by

$$Pr(T_c \leq t) = \frac{1}{|S|} \sum_{s \in S} f(t, n_{d,s}, \mu_{d,s}, \frac{\lambda_c}{|S|}, q_s). \tag{2}$$

The above model does not explicitly account for the replications of objects. As explained in Section 2, the Spotify storage system replicates each object three times. The AP routes a request to the primary server of the object. A different server is only contacted, if the primary server either fails or does not store the object. Since both probabilities are small, we consider in our model only the primary server for an object.

3.3 Evaluation on the lab testbed

The evaluation is performed on our KTH lab testbed, which comprises some 60 rack-based servers interconnected by Ethernet switches. We use two types of servers, which we also refer to as small servers and large servers (Table 1).

3.3.1 Single storage server

In this series of experiments, we measure the response times of a single server having a single disk as a function of the request rate, and we estimate the model parameters. This allows us to validate our analytical model of a single server, as expressed in Equation 1. The requests are generated using a Spotify request trace, and they follow a Poisson arrival process. A request retrieves an object, which has an average size of about 80 KB.

The setup consists of two physical servers, a load injector and a storage server, as shown in Fig. 4 (top). Both servers have identical hardware; they are either small or large servers. All servers run Ubuntu 10.04 LTS. The load injector has installed a customized version of HTTPerf [6]. It takes as input the Spotify trace, generates a stream of HTTP requests, and measures the response times. The storage server runs Nginx [3], an open-source HTTP server.

Before conducting an experiment, we populate the disk of the storage server with objects. A small server is populated with about 280K objects (using 22GB), a large one with about 4,000K objects (using 350GB). Each run of an experiment includes a warm-up phase, followed by a measurement phase. During warm up, the memory is populated with the objective to achieve the same cache hit ratio as the server would achieve in steady state, i.e., after a long period of operation. The memory of the small server holds some 12K objects, the memory of the large server some 750K objects. We use a Spotify request trace for the warmup, with 30K requests for the small server and 1,000K requests for the large server. The measurement runs are performed with a different part of the Spotify trace than the warmup runs. A measurement run includes 30K requests for the small server and 100K requests for the large server.

We perform a series of runs. The runs start at a request rate of 10 and end at 70, with increments of 5, for the small server; they start at a rate of 60

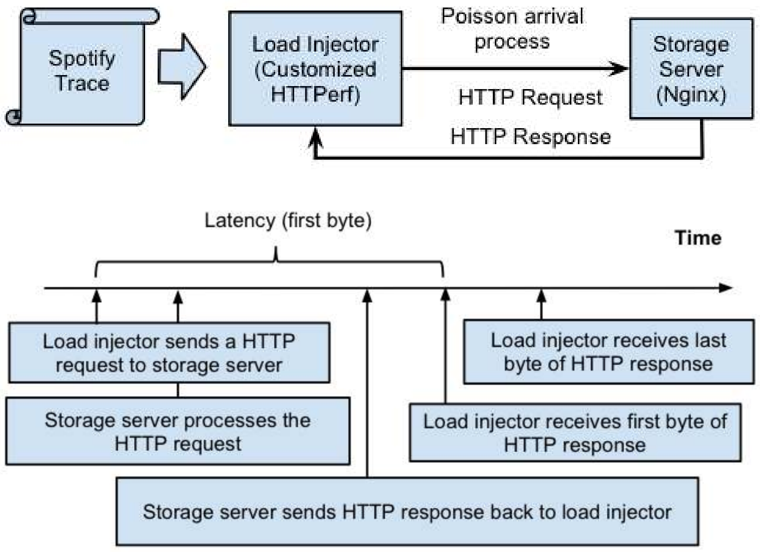

Spotify Trace
Load Injector (Customized HTTPerf)
Poisson arrival process
HTTP Request
HTTP Response
Storage Server (Nginx)
Latency (first byte)
Time
Load injector sends a HTTP request to storage server
Storage server processes the HTTP request
Storage server sends HTTP response back to load injector
Load injector receives first byte of HTTP response
Load injector receives last byte of HTTP response

Fig. 4 Setup and measurement of request latency for a single server.

and end at 140, with increments of 20, for the large server. The response time for a request is measured as indicated in Fig. 4 (bottom). Fig. 6(a) and Fig. 7(a) show the measurement results for three selected latencies for the small and large server, respectively. The vertical axis gives the fraction of requests that have been served within a particular latency. The horizontal axis gives the rate of the request arrival process. All measurement points in a figure that correspond to the same request rate result from a single run of an experiment.

The figures further include the model predictions in the form of solid lines. These predictions come from the evaluation of Equation 1 and an estimation of the model parameters/confidence limits, which will be discussed in Section 3.5.

We make three observations regarding the measurement results. First, the fraction of requests that can be served below a given time decreases as the load increases. For small delays, the relationship is almost linear. Second, the model predictions agree well with the measurements below the confidence limit. In fact, measurements and models diverge at most 11%. A third observation can not be made from the figures but from the measurement data. As expected, the variance of the response times is small for low request rates and becomes larger with increasing rate. For instance, for the small server, we did not measure any response times above 200 msec under a rate of 30; however, at the rate of 70, we measured several response times above 1 sec.

3.3.2 A cluster of storage servers

In this series of experiments, we measure the response times of a cluster of storage servers as a function of the request rate, and we estimate the model parameters. This allows us to validate our analytical model of a cluster as

Small server	Specification
Model	Dell Power edge 750 1U server
RAM	1GB
CPU	Intel(R) Pentium(R) 4 CPU 2.80GHz
Harddisk	Single disk 40GB
Network Controller	Intel 82547GI Gigabit Ethernet Controller

Large server	Specification
Model	Dell PowerEdge R715 2U Rack Server
RAM	64GB
CPU	two 12-core AMD Opteron(tm) processors
Harddisk	Single disk - 500GB
Network Controller	Broadcom 5709C Gigabit NICs

Table 1 Servers on the KTH testbed

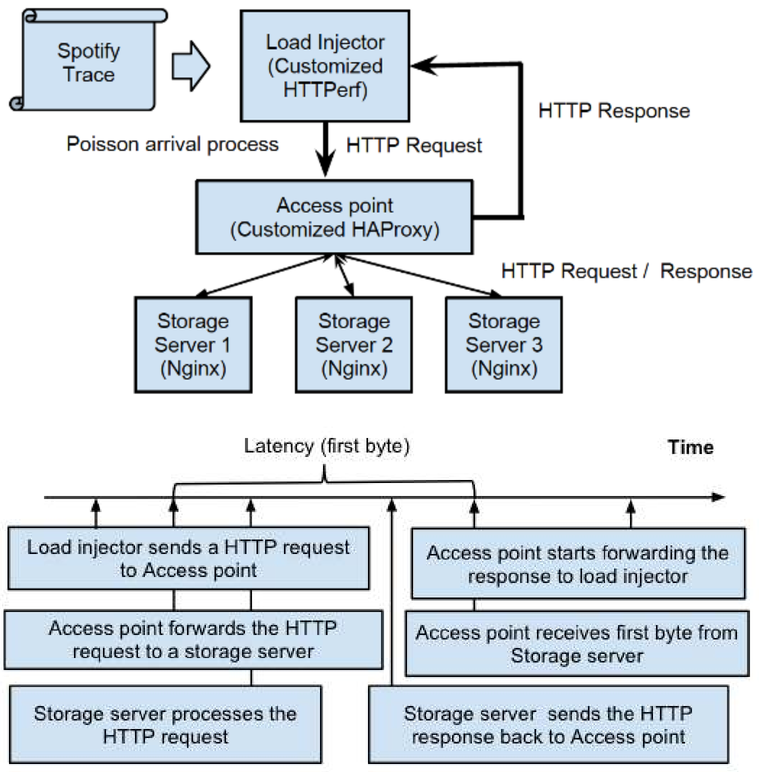
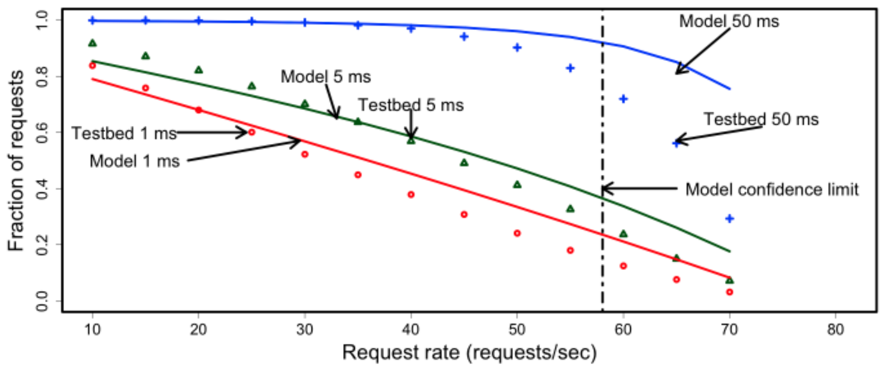

Spotify Trace
Load Injector (Customized HTTPerf)
HTTP Response
Poisson arrival process
HTTP Request
Access point (Customized HAProxy)
HTTP Request / Response
Storage Server 1 (Nginx)
Storage Server 2 (Nginx)
Storage Server 3 (Nginx)
Latency (first byte)
Time
Load injector sends a HTTP request to Access point
Access point starts forwarding the response to load injector
Access point forwards the HTTP request to a storage server
Access point receives first byte from Storage server
Storage server processes the HTTP request
Storage server sends the HTTP response back to Access point

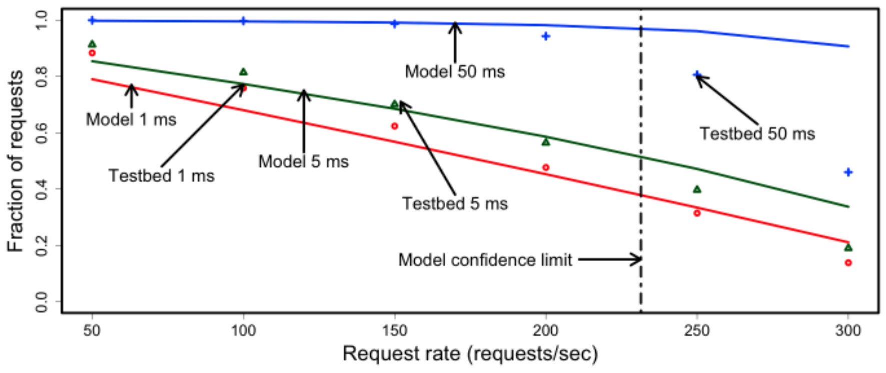
Fig. 5 Setup and measurement of request latency for clusters.

expressed in Equation 2. Similar to the single-server experiments, the requests are generated using a Spotify request trace, and they follow a Poisson arrival process.

We perform experiments for two clusters: one cluster with small servers (one load injector, one AP, and five storage servers) and one with large servers (one load injector, one AP, and three storage servers). The testbed setup can be seen in Fig. 5 (top). The software setup of the load injector and storage servers



(a) Single small server

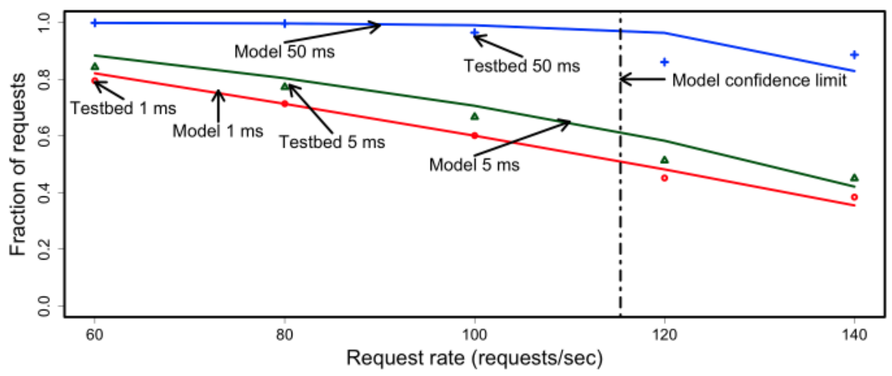


(b) Cluster of small servers

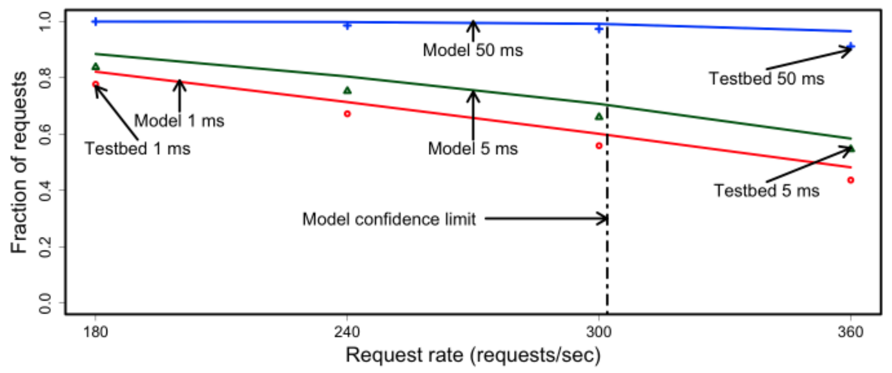
Fig. 6 Lab testbed measurements and model predictions for small servers.

have been discussed above. The AP runs a customized version of HAProxy [7] that forwards a request to a storage server and returns the response to the load injector.

Before conducting an experiment, we populate the disks of the storage servers with objects: we allocate each object uniformly at random to one of the servers. We then create an allocation table for request routing that is placed in the AP. This setup leads to a system whose statistical behavior closely approximates that of the Spotify storage system. During the warmup phase for each run, we populate the memory of all storage servers in the same way as discussed above for a single server. After the warmup phase, the measurement run is performed. Driven by a Spotify trace, the load injector sends a request stream to the AP. Receiving a request from the load injector, the AP forwards it to a storage server according to the allocation table. The storage server processes the request and sends a response to the AP, which forwards it to the load injector. The response time of each request is measured as shown in Fig. 5 (bottom). Regarding dimensioning, the number of allocated objects per server is similar to the one in the experiments discussed above involving single servers. The same is true regarding the number of objects cached in memory,



(a) Single large server



(b) Cluster of large servers

Fig. 7 Lab testbed measurements and model predictions for large servers.

the number of requests for a warmup run, and the number of requests for a measurement run.

For each experimental run, a request stream is generated at a certain rate, and, for each request, the response time is measured. The runs start at a request rate of 50 and end at 300, with increments of 50, for the cluster of small servers; they start at a rate of 180 and end at 360, with increments of 60, for the cluster of large servers. Fig. 6(b) and Fig. 7(b) show the measurement results for three selected latencies for the cluster of small and large servers, respectively. The figures further include the model predictions in the form of solid lines. The predictions are obtained from Equation 2 and model parameters, discussed in Section 3.5. Our conclusions from the experiments on the two clusters are similar to those on the single servers: the fraction of requests that can be served under a given time decreases as the load increases. The relationship is almost linear; the slopes of the curves decrease slightly with increasing request rate. Further, the measurements and models diverge at most 9.3% below the confidence limit.

3.4 Evaluation on the Spotify operational environment

For this evaluation, we had access to hardware and direct, anonymized measurements from the Spotify operational environment. The single server evaluation has been performed on a Spotify storage server, and the cluster evaluation has been performed with measurement data from the Stockholm backend site.

3.4.1 Single storage server

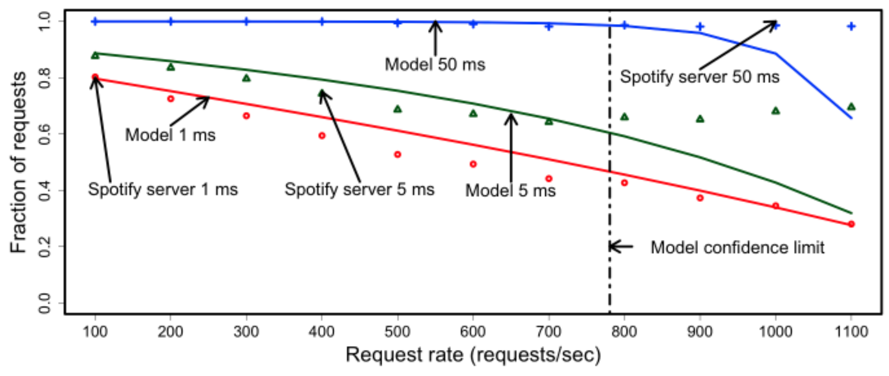
We benchmark an operational Spotify server with the same method as discussed in Section 3.3.1. Such a server stores about 7.5M objects (using 600GB), and a cache after the warm-up phase contains about 375K objects (using 30GB). (The actual capacity of the Spotify server is significantly larger. We only populate 600GB of space, since the traces for our experiment contains requests for objects with a total size of 600GB.) The server has six identical disks and objects are uniformly at random allocated to disks. For a run of the experiment, 1000K requests are processed during the warm-up phase, and 300K requests during the measurement phase. The runs start at a request rate of 100 and end at 1,100, with increments of 100. Fig. 8(a) shows the measurement results for three selected latencies for the Spotify operational server.

The qualitative observations we made for the two servers on the KTH testbed (Section 3.3.1) hold also for the measurements from the Spotify server. Specifically, the measurements and model predictions diverge at most 8.45%, for request rates lower than the model confidence limit.

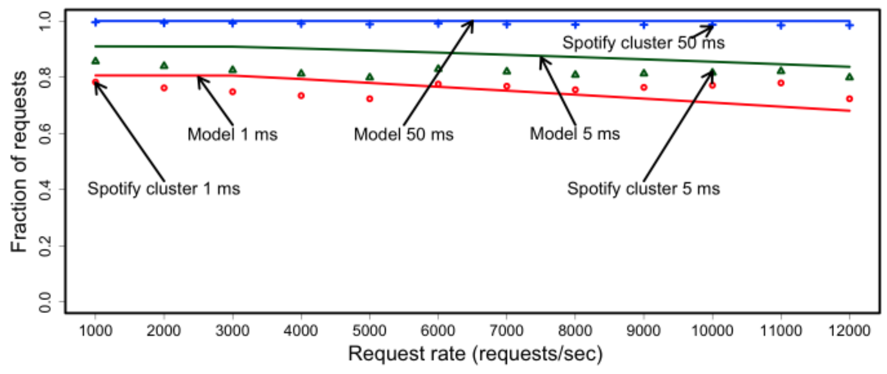
3.4.2 Spotify storage system

For the evaluation, we use 24 hours of anonymized monitoring data from the Stockholm site. This site has 31 operational storage servers. The monitoring data includes, for each storage server, measurements of the arrival rate and response time distribution for the requests that have been sent by the APs. The measurement values are five-minute averages. The data includes also measurements from requests that have been forwarded to the Master Storage, but as stated in Section 3, such requests are rare, below 1% of all requests sent to the storage servers.

Some of the servers at the Stockholm site have a slightly different configuration from the one discussed above. These differences have been taken in account for the estimation of model parameters. Fig. 8(b) presents the measurement results in the same form as those we obtained from the KTH testbed. It allows us to compare the performance of the storage system with predictions from the analytical model. Specifically, it shows measurement results and model predictions for three selected latencies, starting at a request rate of 1,000 and ending at 12,000, with increments of 1,000. The confidence limit is outside the measurement interval, which means that we have confidence that our analytical model is applicable within the complete range of available measurements.



(a) Single Spotify Storage server



(b) Cluster of Spotify Storage servers

Fig. 8 Spotify operational environment measurements and model predictions

We make two observations, considering Fig. 8(b). First, similar to the evaluations we performed on the KTH testbed, the measurements and the model predictions diverge at most 9.61%. This is somewhat surprising, since this operational environment is much more complex and less controllable for us than the lab testbed. For instance, for our testbed measurements, (1) we generate requests with Poisson arrival characteristics, which only approximates arrivals in the operational system; (2) on the testbed we use identical servers, while the production system has some variations in the server configuration; (3) the testbed configurations do not consider Master Storage, etc.

Furthermore, the measurements suggested that the fraction of requests under a specific latency stays almost constant within the range of request rates measured. In fact, our model predicts that, the fraction of requests served within 50 msec stays almost constant until the confidence limit, at about 22,000 requests/sec. Therefore, we expect that this site can handle a much higher load than observed during our measurement period, without experiencing a significant decrease in performance when considering the 50 msec response-time limit. A response time of up to 50 msec provides the user experience that a selected track starts “instantly”.

Parameter	Small server	Large server	Spotify server
μ_d	93	120	150
n_d	1	1	6
γ	0.0137	0.00580	0.000501
q_0	0.946	1.15	0.815

Table 2 Model parameters for a single storage server

3.5 Estimating model parameters / confidence limit

We determine the model parameters for the single server, given in Equation 1, namely, the service rate of a disk μ_d, the number of identical disks n_d, and the probability that a request is served from memory q. While n_d can be read out from the system configuration, the other two parameters are obtained through benchmarking. We first estimate the average service time T_s of the single disk through running iostat [8] while the server is in operation (i.e. after the warm-up phase), and we obtain $\mu_d = 1/T_s$. We estimate parameter q as a fraction of requests that have a latency below 1 msec while the server is in operation. Fig. 9 shows the measured values for q for different server types and request rates. We observe a significant difference in parameter q between the testbed servers (small and large server) and a Spotify operational server. We believe that this difference is due to the fact that software and hardware of the Spotify server is highly optimized for Spotify load, while the testbed servers are general-purpose machines and configured with default options.

We observe in Fig. 9 that, for all three server types, the value of q decreases linearly with increasing request rate. This behavior can be explained by the server software, which includes Nginx on a Linux distribution. Nginx employs an event-driven model for handling concurrent requests that are sent to a configurable number of processes k. Such a process, accessing disk I/O in the installed version of Linux, may be blocked and put into an uninteruptible-sleep state [9]. (There exists an asynchronous implementation of Linux disk I/O [10], which, however, does not support disk caching in memory and, therefore, is not applicable to our case.) As a result, when all processes are blocked, a newly arrived request that can be served from memory must wait for a process to become unblocked. As the probability that all processes are blocked at the same time increases (roughly) linearly with the request rate, the probability that a newly arrived request can be served from memory without waiting decreases similarly. We approximate, through least-square regression, q with the linear function $q = -\gamma\lambda + q_0$, whereby λ is the request rate. All model parameters of a single storage server are summarized in Table 2.

Fig. 9 leads us to a modified interpretation of the parameter q in the model described in Section 3.1: q is the probability that an arriving request (a) can be served from memory and (b) there are at most $k-1$ disk requests currently executing.

The model for a cluster, given by Equation 2, contains the model parameters of each server in the cluster. Therefore, to estimate the parameters of the cluster, the parameters of each individual server need to be obtained.

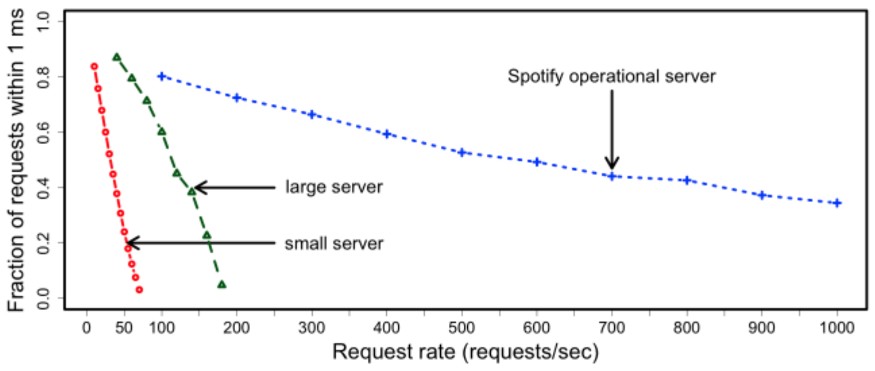


Fig. 9 Estimating the parameter q

We now compute the *model confidence limit for a single server*, i.e., the maximum request rate up to which we feel confident that our model (i.e., Equation 1) applies. Through extensive testing, we found that our model predictions are close to the measurements from the real system, as long as the average length of any disk queue is at most one. Note that, for a queue length of at most one, the estimation errors considering the mean service time (which refers to the inverse of the service rate of a disk μ_d), and the actual service time distribution (which our model assumes to be exponential) do not have significant effects on observed system performance. To increase the confidence limit, both μ_d and the service time distribution should be estimated with higher precision. We attempted to estimate the service time distribution with higher accuracy and failed after many tries.

From the behavior of an M/M/1 queue we know that the average queue length for each disk is $L_d = \frac{\lambda_d}{\mu_d - \lambda_d}$. Applying the linear approximation for q and setting $L_d = 1$, simple manipulations give the model confidence limit λ_L for a single server as the positive root of $\gamma\lambda_L^2 + (1 - q_0)\lambda_L - \frac{1}{2}\mu_d n_d = 0$. The confidence limits in Fig. 6(a), Fig. 7(a), and Fig. 8(a) are computed using this method. As can be observed from the figures, increasing request rates beyond the confidence limits coincides with a growing gap between model predictions and measurements, specifically for the latency of 50 msec, which is an important value for the Spotify storage system.

We now discuss the *model confidence limit for the cluster*, i.e., the maximum request rate up to which each individual server is loaded within its specific confidence limit.

The allocation of objects to primary servers in the Spotify storage system can be approximated by a process whereby each object is placed on a server uniformly at random, weighted by the storage capacity of the server. (In Section 4, we refer to this allocation policy as the random policy.) Therefore, the number of objects allocated to a server can vary, even for homogeneous servers. To compute the confidence limit for the cluster, we must know the load of the highest loaded server, since the load of each server must be below its confidence limit.

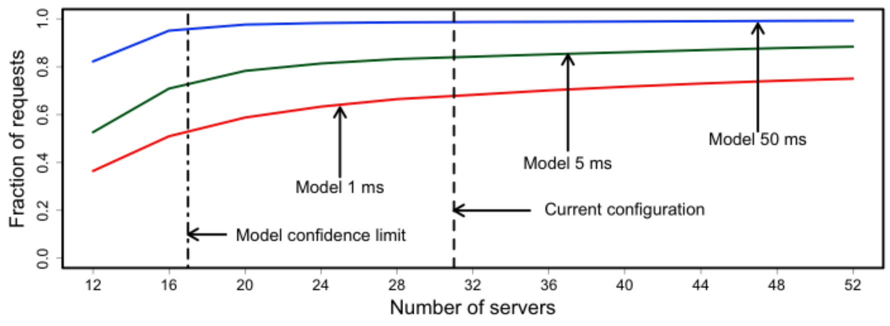
If the server contains a large number of objects, as in our system, the expected load on the server is (approximately) proportional to the number of objects. The distribution of the number of objects across servers, and, therefore, the distribution of the load, can be modeled using the balls-and-bins model [11]. We interpret balls as objects or requests and bins as servers. A result from the analysis of the balls-and-bins model states that, when m balls are thrown independently and uniformly at random into n bins and $m \gg n \cdot (\log n)$ can be assumed, then, with high probability, there is no bin that receives more than $M = \frac{m}{n} + \sqrt{\frac{2m \log n}{n}}$ balls [12]. Applying this result to our case, we can state that there is no server with load above M, with high probability. (The balls-and-bins model applies in our case, because (a) objects are allocated according to the random policy and (b) request rates are more than 4000 requests/sec on average on a Spotify backend site, while the number of servers is at most 50, as described in Section 3.4.)

Using the above result, we derive the confidence limit $\lambda_{L,c}$ of the cluster c as the smaller root of $\frac{1}{|S|^2}\lambda_{L,c}^2 + (-\frac{2\lambda_L}{|S|} - \frac{2\log|S|}{|S|})\lambda_{L,c} + \lambda_L^2 = 0$, where λ_L is the minimum of the confidence limits of all the servers. The confidence limits in Fig. 6(b), Fig. 7(b), and Fig. 8(b) are computed using this method. As for the single server, the model predictions for the cluster diverge significantly from the measurements for rates beyond the confidence limits.

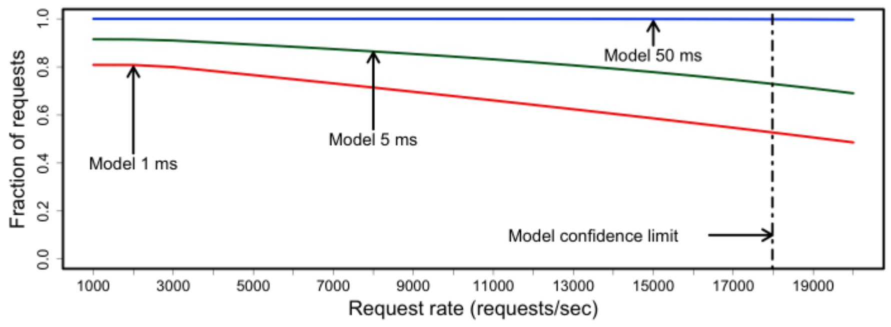
3.6 Applications of the model

We apply the analytical model to predict, for the Spotify storage system at the Stockholm site, the fraction of requests served under given latencies for a load of 12,000 requests/sec, which is the peak load from the dataset we used. While our evaluation has involved 31 servers, we use the model to estimate response time for configurations from 12 to 52 storage servers. The result is shown in Fig. 10a. The confidence limit is 17 servers. Above this number, we have confidence that the model applies. We observe that for each latency curve in the figure the slope decreases with increasing number of servers. This means that adding additional servers to the storage system of, say, 20 servers result in a much larger reduction of response time than adding servers to the storage system of, say, 50 servers. Further, we can see that the quality-of-service requirement has a significant effect on the required number of servers, and substantial investment would be needed to increase the service quality. For example, if the quality-of-service requirement changes from serving 80% of the requests within 5 msec to either (1) serving 80% of the requests within 1 msec, or (2) serving 90% of the requests within 5 msec, the number of required servers must increase from less than 24 to ca. 52.

Second, we predict the fraction of requests served under specific response times for a storage system with 25 servers. We consider a scenario where the load varies from 1,000 to 20,000 requests/sec. The result is shown in Fig. 10b. The confidence limit is 18,000 below which our model applies. We observe that the slope of all curves in the figure is almost zero between 1,000 to 3,000



(a) Varying the number of servers in the storage system for a load of 12,000 requests/sec



(b) Varying the load in the storage system for 25 storage servers

Fig. 10 Applications of the model to system dimensioning

requests/sec, beyond which it starts decreasing. We can predict that increasing the load on the storage system from 1,000 to 3,000 requests/sec does not have any measurable impact on performance, while we expect that an increase from 1,000 to 15,000 requests/sec clearly will. Our model also predicts that, for a response time limit of 50 msec, the fraction of requests served within the response time limit remains almost unchanged for rates between 1,000 and 18,000 requests/sec.

4 Estimating the capacity of a storage cluster for different object allocation policies

A Spotify backend site (see Fig. 2) stores more than 60 million objects, which makes complete replication of objects across servers impractical. Therefore, each object is stored on, or allocated to, a subset of servers. In the Spotify storage system, each object is allocated to three servers. For the discussion in this section, we can assume that each object is allocated to a single server only, because the two other replicas are only accessed in rare cases, including failure of the primary server or cache miss.

In this section, we develop and validate a model for the capacity of a storage cluster using two different allocation policies. An *object allocation policy* defines the allocation of a set of objects to a set of servers.

The first policy, which we call the *random policy*, allocates an object uniformly at random to a server in the cluster. The second policy, which we call the *popularity-aware policy*, allocates an object to a server depending on how frequently the object is requested, i.e. the request rate for this object. The allocation is performed in such a way that each server experiences the same (or almost the same) aggregate request rate.

We call the aggregate request rate to a server also the *load* on the server, and, similarly, we call the aggregate request rate to a cluster the load on the cluster. We denote the maximum request rate that a server can process without violating a local service objective the *capacity* of the server. (In this work, we consider an objective that the 95^{th} percentile of the response time to a request is below 50 msec.) Similarly, by the capacity of the cluster, we understand the maximum request rate to a cluster, such that each server serves a rate below its capacity.

Assuming the request rate for each object is known, the popularity-aware policy is optimal (for a cluster of homogeneous servers) in the sense that it results in the largest possible capacity of the cluster, since all servers carry the same load. In contrast, under the random policy, servers experience different request rates, and, therefore, the cluster capacity is lower than under the popularity-aware policy. The model we develop in this section will quantify the difference in capacity between the two policies. While the popularity-aware policy is generally superior in performance, the random policy is less complex to implement. The popularity-aware policy requires a (large) routing table, which must be maintained, and the object allocation needs to be adapted as the popularity of objects changes. On the other hand, routing under the random policy can be realized with functions that approximate random assignment, e.g. hash functions, and the allocation does not need to be adapted when the popularity of objects changes. The Spotify backend control system uses a random allocation policy for reasons that will become clear in this section.

4.1 The load for a server under the random policy

Let L_i be a random variable representing the request rate seen by a specific object i on a given server. Assuming that objects $1..n$ are allocated to this server, the load on the server is $L = \sum_{i=1}^{n} L_i$. Fig. 11 shows the distribution of the object request rate for a Spotify backend site, computed from a one-day trace. The figure also shows a Pareto distribution (scale parameter $x_{min} = 0.33$, shape parameter $\alpha = 1.55$), obtained through a maximum-likelihood estimator, which approximates this data. (We evaluated the goodness-of-fit of the Pareto distribution through a Kolmogorov-Smirnov(KS) test [13], which measures the distance between the empirical distribution and the Pareto distribution. Using 1000 samples, we obtained a distance 0.0116, which is a good fit according to [14].) In the following, we use the Pareto distribution for L_i.

From [15], Equation 28, we find that the (1-q)-quantile of the sum of n i.i.d. Pareto random variables with shape parameter α, for large n, can be written as

$$z_{1-q}(n) \approx n^{1/\alpha}(1-q)^{-1/\alpha} + \frac{n\alpha}{\alpha-1} \text{ for } 1 < \alpha < 2.$$

If we let q go to 0, we obtain an approximation of the 1-quantile $z_1(n)$. Since we can interpret $z_1(n)$ as the load L on a server with n objects, we get

$$L \approx n^{1/\alpha} + \frac{n\alpha}{\alpha-1} \text{ for } 1 < \alpha < 2. \tag{3}$$

4.2 The capacity of a cluster under random and popularity-aware policies

Assume the cluster contains a set S of homogeneous storage servers (see Fig. 2). Each server has the capacity of Ω requests/sec, that is, the maximum request rate for which the service objectives are not violated. We define Ω_c, the capacity of the cluster under arbitrary allocation policy as the maximum request rate for which the service objectives are not violated at any of the servers. We assume that the access point forwards a request independently to a server s with probability p_s. Then, the capacity of the cluster is determined by $p_m = max_{s \in S}\{p_s\}$ as follows

$$\Omega_c = \frac{\Omega}{p_m} \; . \tag{4}$$

As mentioned before, under the popularity-aware policy, all servers experience the same (or almost the same) load. As a result, $p_s = \frac{1}{|S|} \; \forall s \in S$, and the capacity of the cluster under the popularity-aware policy, denoted by $\Omega_c(popularity)$, becomes

$$\Omega_c(popularity) = \Omega \cdot |S| \; . \tag{5}$$

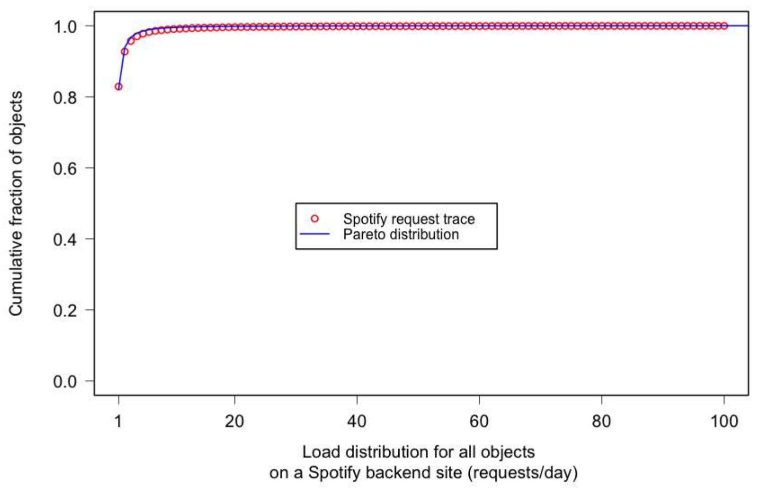
In order to compute the capacity of the cluster under the random policy, we estimate the maximum number of objects on any server of the cluster. The estimate is obtained from an analysis of the balls-and-bins model, which has been discussed in Section 3.5. Following this analysis and assuming O is the set of all objects and O_m the set of objects allocated to server m, i.e., the server with the highest load, we can write

$$|O_m| \approx \frac{|O|}{|S|} + \sqrt{\frac{2|O|\log|S|}{|S|}} \tag{6}$$

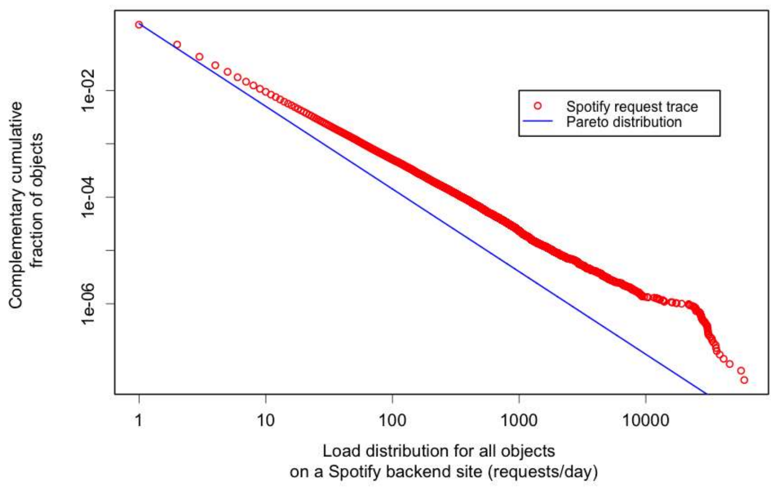
if $|O| \gg |S|(\log|S|)$.

Using Equation 3, we get for the maximum load of any server in the cluster

$$L_m = z_1(|O_m|) \approx |O_m|^{1/\alpha} + \frac{|O_m|\alpha}{\alpha-1} \; . \tag{7}$$



(a)



(b)

Fig. 11 Approximating the load distribution using a Pareto distribution with shape parameter $\alpha = 1.55$ and scale parameter $x_{min} = 0.33$.

And for p_m,

$$p_m \approx \frac{z_1(|O_m|)}{z_1(|O|)} = \frac{|O_m|^{1/\alpha} + \frac{|O_m|\alpha}{\alpha-1}}{|O|^{1/\alpha} + \frac{|O|\alpha}{\alpha-1}} \ . \tag{8}$$

Applying Equation 4, the cluster capacity for the random policy, denoted by $\Omega_c(random)$, becomes

$$\Omega_c(random) \approx \Omega \cdot \frac{|O|^{1/\alpha} + \frac{|O|\alpha}{\alpha-1}}{|O_m|^{1/\alpha} + \frac{|O_m|\alpha}{\alpha-1}} \tag{9}$$

assuming $|O| \gg |S|(\log|S|)$. Recall that Equation 7, 8, and 9 assume that $1 < \alpha < 2$.

Note that, while the condition $|O| \gg |S|(\log|S|)$ holds for a Spotify backend site, the discussion in this section applies generally to information systems whose object popularity follows a Pareto distribution and where the above condition is true.

We now compare the performance of the popularity-aware policy with that of the random policy, by computing the *relative cluster capacity*, namely $\frac{\Omega_c(random)}{\Omega_c(popularity)}$, in function of the number of objects $|O|$ in the system and the number of servers $|S|$ in the cluster. Fig. 12(a) and Fig. 12(b) provide this comparison, based on which we make three observations.

First, for a fixed number of objects, the relative capacity decreases (sublinearly) with the number of servers. For example, for 60,000 objects, the relative capacity is about 0.80 for 200 servers, while the relative capacity decreases to about 0.63 for some 1,000 servers. This is expected, because an increase in the number of servers results in a higher variation of number of objects allocated to a server, and, therefore, an increasing difference in capacity between the two policies. Second, for a fixed number of servers, the relative capacity increases with the number of objects.

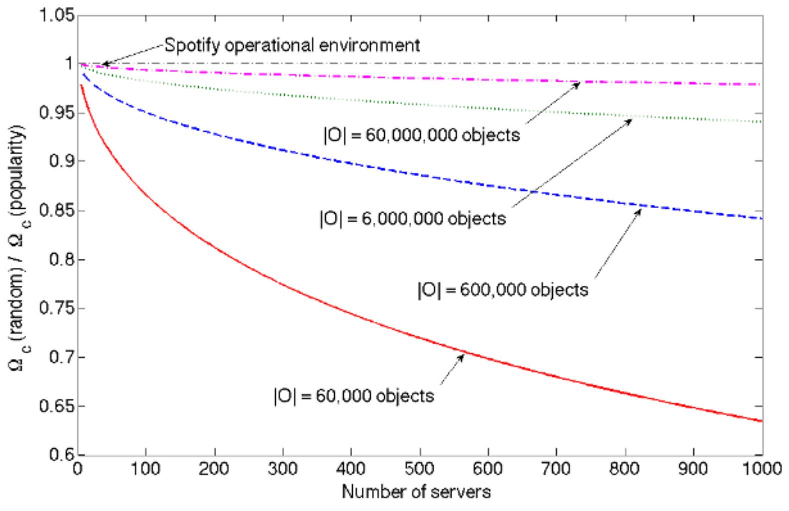
Third, and most importantly, the random policy results in a cluster capacity that approaches that of the popularity-aware policy, if $|O|/|S|\log(|S|)$ is large. In other words, for a system with a large value of $|O|/|S|\log(|S|)$, adopting either policy does not make a significant difference. This is exactly the case for the Spotify storage system, as can be seen from Fig. 12(a), and, therefore, the additional complexity of implementing the popularity-aware policy in the Spotify backend control system is not needed.

4.3 Evaluation of the model on the lab testbed

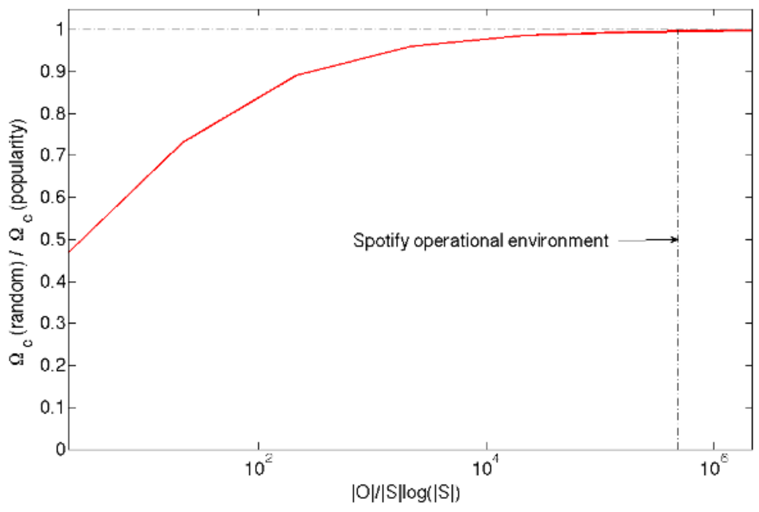
In this series of experiments, we measure the cluster capacity for different number of objects under random and popularity-aware policies. This allows us to validate our analytical model of a cluster, as expressed in Equations 5 and 9.

For experimentation, we use the cluster of five small servers described in Section 3.3.2. From the measurement results presented in Section 3.3.1 and Fig. 6(a), we infer that the capacity of a single server in this cluster Ω is 40 requests/sec, which corresponds to the maximum rate at which the server serves 95% of requests within 50 msec.

Each experimental run is performed for a specific number of objects, which are 1250, 1750, 2500, 5000, and 10000. For each number of objects, we evaluate



(a)



(b)

Fig. 12 Relative cluster capacity of the random allocation policy versus the popularity-aware policy in function of (a) the number of servers and (b) the number of objects.

the system for the random and for the popularity-aware policy. Also, for each number of objects k, we create a request trace from the Spotify trace, by selecting k objects uniformly at random and sampling from the load distribution for objects (Fig. 11) to obtain a request trace of 100,000 requests.

For each experimental run, a request stream is generated at a certain rate as described in Section 3.3.2. During a run, we periodically measure the number

Number of	Random policy			Popularity-aware policy		
objects	Measurement	Model	Error(%)	Measurement	Model	Error(%)
1250	166.50	176.21	5.83	190.10	200	5.21
1750	180.30	180.92	0.35	191.00	200	4.71
2500	189.70	182.30	3.90	190.80	200	4.82
5000	188.40	186.92	0.78	191.00	200	4.71
10000	188.60	190.39	0.95	192.40	200	3.95

Table 3 Lab testbed measurements and model predictions of model for estimating a cluster capacity. (All capacities are in requests/sec.)

Number of	Relative cluster capacity		
objects	Measurement	Model	Error(%)
1250	0.88	0.88	0
1750	0.94	0.90	4.17
2500	0.99	0.91	8.32
5000	0.99	0.93	5.25
10000	0.98	0.95	2.89

Table 4 Lab testbed measurements and model predictions of model for estimating a relative cluster capacity.

of object requests forwarded to each server using Zabbix agents, which allows us to estimate the request rate to each server. We adjust the request rate to the cluster such that the maximum request rate for all storage servers is within 1% from $\Omega = 40$ requests/sec, and report this rate as the cluster capacity.

Tables 3 and 4 contain the results of the measurements. For system configurations with certain number of objects and both allocation policies, Table 3 shows the measured cluster capacities, the model predictions, and the prediction errors. Table 4 shows the relative cluster capacities obtained from measurements, the model predictions, and the prediction errors.

We make two observations from the measurement results. First, within the parameter range of the measurements, the capacity of the cluster under the popularity-aware policy does not depend on the number of objects. On the other hand, the capacity of the cluster under the random policy increases with the number of objects, as expected from our model. Second, the model predictions are close to the measurements, with the error of at most 8.32%. We can conclude that the testbed cluster performs according to the model predictions, which contributes to the validation of the model.

5 Related work

Extensive research has been undertaken in modeling the performance of storage devices [16–21]. The authors in [16] present a performance model for Ursa Minor[22], a robust distributed storage system. The model allows them to predict the average latency of a request, as well as the capacity of the system. Measurement-based performance models of storage systems are presented in [17,18]. In [17], expected latency and throughput metrics are predicted for different allocation schemes of virtual disks to physical storage devices. The authors of [18] predict the average response time of I/O requests when multiple virtual machines are consolidated on a single server, while in [21] architectures for clustered video servers are proposed.

The development and evaluation of distributed key-value stores is an active research area. In contrast to Spotify's storage system design, which is hierarchical, many advanced key-value storage systems in operation today are based on a peer-to-peer architecture. Among them are Amazon Dynamo [23], Cassandra [24], and Scalaris [25]. Facebook's Haystack storage system follows a different design, which is closer to Spotify's. Most of these systems use some form of consistent hashing to allocate objects to servers. The differences in their designs are motivated by their respective operational requirements, and they relate to the number of objects to be hosted, the size of the objects, the rate of update, the number of clients, and the expected scaling of the load. While these systems generally provide more functionality than the Spotify storage system does, to our knowledge, no performance model has been developed for any of them.

Hierarchical caching systems share similarities with our architecture, and many works have studied and modeled the performance of such systems [26–32]. Most of these works model the 'cache hit ratio', or the fraction of requests that can be served from caches, for different caching levels and replacement algorithms, e.g., Least Recently Used (LRU) or Least Frequently Used (LFU). A few also model the distribution of response times [26,30], but (1) the models are in complex forms, for which approximations are provided, and (2) the models are evaluated using simulation only. In contrast, our work focuses on modeling the distribution of response times in closed form, and our models are evaluated both on a testbed and in an operational environment.

The modeling of the distribution of response times is addressed in [19] for a single storage server and is based on a two-stage queuing network. In [20], a queuing model is developed to model response times of a cluster of servers, considering general service times. The model gives the response time in a transformed form, or by its moments, and further approximations would be needed to derive its distribution. In [21] the service time distribution in clustered video servers is evaluated, approximating non-dominant delay components with constant values. Note that the models in [19] and [20] are evaluated on a simulator only, and the works do not include testbed experiments.

Compared to the above results, this paper considers a complete distributed system system. It predicts the distribution of the response times through a simplified system model, and justifies the system model thorough experimental evaluation.

Automated control frameworks for dynamically adjusting the number of storage servers in a stateful storage system (where a specific request can be served from only a subset of servers) recently gained popularity. Some frameworks, e.g., SCADS[33] and ElastMan [34], are designed to achieve a target response time expressed in quantiles, while others, e.g., Yak [35], target an average response time. While our paper does not address the dynamic configuration of a storage system, it complements previous works by proposing an initial operating point, which can be subsequently adapted using a control framework.

Object placement for networked multimedia storage system is a well-studied area. Most works deal with video distribution systems. Many of them evaluate the benefits of a popularity-aware allocation policy over other policies, for instance, [36–39]. These works all find that a popularity-aware allocation policy provides higher throughput than other policies. Our work is complementary to the previous efforts, as it investigates the system dimensioning and configuration parameters, for which either a popularity-aware or a random policy provides better performance. In our context, we show that the key parameters are the number of objects and the number of servers in the system.

6 Conclusion

We make the following contributions with this paper. First, we introduce an architectural model of a distributed key-value store that simplifies the architecture and functionality of the Spotify storage system. Second, we present a queuing model that allows us to compute the response time distribution of the storage system. Then, we perform an extensive evaluation of the model, first on our testbed and later on the Spotify operational infrastructure. This evaluation shows that the model predictions are accurate, with an error of at most 11%. Third, we present an analytical model for estimating the capacity of a storage cluster under the random and popularity-aware object allocation policies. Measurements on our testbed are within 9% of the model predictions. For the case of a Spotify backend site, the model suggests that both policies result in (approximately) the same performance. This justifies the Spotify design, which adopts the random policy (which is easier to implement than the popularity-aware policy).

The reported errors result from the fact that we use a simple model to describe, on an abstract level, the behavior of a complex distributed system. This simplicity is a virtue insofar as it provides better insight and can be computed in real time. The downside, in the case of the model for response time distribution, is that the applicability of the model is restricted to a lightly loaded system; however, this corresponds to the operational range of the Spotify storage system (see further discussion below). To increase the accuracy of the model, or to extend the range of load patterns for which it can make predictions, one needs to refine the system model. Such refinements can include, modeling the specific software process structure in a server, the OS scheduling policies, heterogeneous hardware, the detailed request routing policy in the cluster, as well as the real arrival process and service discipline of the queuing model. The difficulty will be to identify refinements that significantly increase the accuracy of the model while keeping it simple and tractable.

Note that the process of developing the (relatively) simple models in this paper has not been as straightforward as it may seem. To the contrary, this work was difficult and lengthy. We started out with studying the Spotify backend architecture in great detail and simplified this complex system in several iterations, in order to make its analysis feasible. During this work, we devised several alternative models, each of which we validated through experimentation. The models in this paper are the results of all this work.

As we have shown, the confidence range of our model covers the entire operational range of the load to the Spotify storage system. As we have validated through experimentation, the performance of the system deteriorates when the load significantly surpass the model-predicted confidence limit. Lastly, applying our model, we predict for a specific Spotify backend site that the system could handle the peak load observed during a specific day with fewer servers, or, alternatively, that the system with 25 servers could handle a significantly higher load than observed, without noticable performance degradation (for important response time limit, which is 50 msec).

This work is important to Spotify, since latency is the key performance metric of its storage system. The main reason for this is that estimating latency distributions and capacity is essential to guarantee the quality of the overall service. Thus, while the system we evaluated is overprovisioned, our models allow for predictions and capacity planning to accommodate user growth and site failures. This growth in Spotify is at times very bursty, e.g., when the service is launched in new countries. Note though that the results in this paper extend beyond Spotify's technology. In fact, our approach can be applied to similar types of distributed key-value stores and other services that rely on them, such as video streaming, as long as the assumptions in Section 3 hold. For instance, depending on the size of the system and the object space, our model can suggest the suitable object allocation policy, either the popularity-aware (if the ratio of the number of objects and the number of servers is small) or the random policy (if the ratio is large, see Equation 9).

As for future work, we plan to develop a subsystem that continuously estimates the model parameters at runtime, taking into account that the resources of the storage servers may be used by other processes than object retrieval, for instance, for maintainence or system reconfiguration. Based on such a capability, we envision an online performance management system for a distributed key-value store like the Spotify storage system.

References

1. Kreitz, G., Niemelä, F.: Spotify – large scale, low latency, P2P music-on-demand streaming. In: Peer-to-Peer Computing, pp. 1–10. IEEE (2010)
2. Yanggratoke, R., Kreitz, G., Goldmann, M., Stadler, R.: Predicting response times for the spotify backend. In: Network and Service Management (CNSM), 2012 8th International Conference on, pp. 117 –125 (2012)
3. Sysoev, I.: Nginx (2002). `http://nginx.org/`
4. Karger, D.R., Lehman, E., Leighton, F.T., Panigrahy, R., Levine, M.S., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: F.T. Leighton, P.W. Shor (eds.) STOC, pp. 654–663. ACM (1997)
5. Kleinrock, L.: Theory, Volume 1, Queueing Systems. Wiley-Interscience (1975)
6. Mosberger, D., Jin, T.: httperf - a tool for measuring web server performance. SIGMETRICS Perform. Eval. Rev. **26**(3), 31–37 (1998)
7. Tarreau, W.: Haproxy. `http://haproxy.1wt.eu/`
8. Godard, S.: iostat. `http://linux.die.net/man/1/iostat`

9. Sovani, K.: Kernel korner - sleeping in the kernel. http://www.linuxjournal.com/article/8144
10. Jones, M.T.: Boost application performance using asynchronous i/o. http://www.ibm.com/developerworks/linux/library/l-async/
11. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press (2005)
12. Raab, M., Steger, A.: "Balls into Bins" - A Simple and Tight Analysis. In: Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science, RANDOM '98, pp. 159–170. Springer-Verlag, London, UK (1998)
13. Massey, F.J.: The kolmogorov-smirnov test for goodness of fit. Journal of the American Statistical Association **46**(253), 68–78 (1951). DOI 10.1080/01621459.1951.10500769. URL http://www.tandfonline.com/doi/abs/10.1080/01621459.1951.10500769
14. Goldstein, M., Morris, S., Yen, G.: Problems with fitting to the power-law distribution. The European Physical Journal B - Condensed Matter and Complex Systems **41**, 255–258 (2004). DOI 10.1140/epjb/e2004-00316-5. URL http://dx.doi.org/10.1140/epjb/e2004-00316-5
15. Zaliapin, I.V., Kagan, Y.Y., Schoenberg, F.P.: Approximating the distribution of pareto sums. Pure and Applied Geophysics **162**, 1187–1228 (2005). 10.1007/s00024-004-2666-3
16. Thereska, E., Abd-El-Malek, M., Wylie, J., Narayanan, D., Ganger, G.: Informed data distribution selection in a self-predicting storage system. In: Autonomic Computing, 2006. ICAC '06. IEEE International Conference on, pp. 187 – 198 (2006). DOI 10.1109/ICAC.2006.1662398
17. Gulati, A., Shanmuganathan, G., Ahmad, I., Waldspurger, C., Uysal, M.: Pesto: online storage performance management in virtualized datacenters. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11, pp. 19:1–19:14. ACM, New York, NY, USA (2011)
18. Kraft, S., Casale, G., Krishnamurthy, D., Greer, D., Kilpatrick, P.: Io performance prediction in consolidated virtualized environments. SIGSOFT Softw. Eng. Notes **36**(5), 295–306 (2011)
19. Xiong, K., Perros, H.: Service performance and analysis in cloud computing. In: IEEE Congress on Services, pp. 693 –700 (2009)
20. Khazaei, H., Misic, J., Misic, V.: Performance analysis of cloud computing centers using m/g/m/m+r queuing systems. IEEE Transactions on Parallel and Distributed Systems **23**(5), 936 –943 (2012)
21. Tewari, R., Mukherjee, R., Dias, D., Vin, H.: Design and performance tradeoffs in clustered video servers. In: Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on, pp. 144–150 (1996). DOI 10.1109/MMCS.1996.534966
22. Abd-El-Malek, M., Courtright II, W.V., Cranor, C., Ganger, G.R., Hendricks, J., Klosterman, A.J., Mesnier, M., Prasad, M., Salmon, B., Sambasivan, R.R., Sinnamohideen, S., Strunk, J.D., Thereska, E., Wachs, M., Wylie, J.J.: Ursa minor: versatile cluster-based storage. In: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4, FAST'05, pp. 5–5. USENIX Association, Berkeley, CA, USA (2005)
23. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**(6), 205–220 (2007)
24. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
25. Schütt, T., Schintke, F., Reinefeld, A.: Scalaris: reliable transactional p2p key/value store. In: Proceedings of the 7th ACM SIGPLAN workshop on ERLANG, ERLANG '08, pp. 41–48. ACM, New York, NY, USA (2008)
26. Che, H., Tung, Y., Wang, Z.: Hierarchical web caching systems: modeling, design and experimental results. Selected Areas in Communications, IEEE Journal on **20**(7), 1305–1314 (2002). DOI 10.1109/JSAC.2002.801752
27. Hou, Y., Pan, J., Li, B., Tang, X., Panwar, S.: Modeling and analysis of an expiration-based hierarchical caching system. In: Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE, vol. 3, pp. 2468–2472 vol.3 (2002). DOI 10.1109/GLOCOM.2002.1189074

28. Hou, Y., Pan, J., Li, B., Panwar, S.: On expiration-based hierarchical caching systems. Selected Areas in Communications, IEEE Journal on **22**(1), 134–150 (2004). DOI 10.1109/JSAC.2003.818804
29. Hu, X., Zincir-Heywood, A.: Understanding the performance of cooperative web caching systems. In: Communication Networks and Services Research Conference, 2005. Proceedings of the 3rd Annual, pp. 183–188 (2005). DOI 10.1109/CNSR.2005.61
30. Fricker, C., Robert, P., Roberts, J.: A versatile and accurate approximation for lru cache performance. In: Proceedings of the 24th International Teletraffic Congress, ITC '12, pp. 8:1–8:8. International Teletraffic Congress (2012). URL `http://dl.acm.org/citation.cfm?id=2414276.2414286`
31. Ho, K.M., Poon, W.F., Lo, K.T.: Investigating the performance of hierarchical video-on-demand system in heterogeneous environment. In: Information Networking, 2008. ICOIN 2008. International Conference on, pp. 1–5 (2008). DOI 10.1109/ICOIN.2008.4472804
32. Karedla, R., Love, J., Wherry, B.: Caching strategies to improve disk system performance. Computer **27**(3), 38–46 (1994). DOI 10.1109/2.268884
33. Trushkowsky, B., Bodík, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: The scads director: scaling a distributed storage system under stringent performance requirements. In: Proceedings of the 9th USENIX conference on File and stroage technologies, FAST'11, pp. 12–12. USENIX Association, Berkeley, CA, USA (2011). URL `http://dl.acm.org/citation.cfm?id=1960475.1960487`
34. Al-Shishtawy, A., Vlassov, V.: Elastman : Autonomic elasticity manager for cloud-based key-value stores. Tech. Rep. 12:01, KTH, Software and Computer Systems, SCS (2012). QC 20120831
35. Klems, M., Silberstein, A., Chen, J., Mortazavi, M., Albert, S.A., Narayan, P., Tumbde, A., Cooper, B.: The yahoo!: cloud datastore load balancer. In: Proceedings of the fourth international workshop on Cloud data management, CloudDB '12, pp. 33–40. ACM, New York, NY, USA (2012). DOI 10.1145/2390021.2390028. URL `http://doi.acm.org/10.1145/2390021.2390028`
36. Dario, V., Cesar, M., Yacine, G.D.: Performance evaluation of an object management policy approach for p2p networks. International Journal of Digital Multimedia Broadcasting **2012** (2012). DOI 10.1155/2012/189325. URL `http://www.hindawi.com/journals/ijdmb/2012/189325/cta/`
37. Fujimoto, T., Endo, R., Matsumoto, K., Shigeno, H.: Video-popularity-based caching scheme for p2p video-on-demand streaming. In: Advanced Information Networking and Applications (AINA), 2011 IEEE International Conference on, pp. 748 –755 (2011). DOI 10.1109/AINA.2011.103
38. Chellouche, S., Negru, D., Chen, Y., Sidibe, M.: Home-box-assisted content delivery network for internet video-on-demand services. In: Computers and Communications (ISCC), 2012 IEEE Symposium on, pp. 000,544 –000,550 (2012). DOI 10.1109/ISCC.2012.6249353
39. Zhou, Y., Fu, T., Chiu, D.M.: Division-of-labor between server and p2p for streaming vod. In: Quality of Service (IWQoS), 2012 IEEE 20th International Workshop on, pp. 1 –9 (2012). DOI 10.1109/IWQoS.2012.6245979

Author Biographies

Rerngvit Yanggratoke is pursuing his Ph.D. under the supervision of Prof. Rolf Stadler at the School of Electrical Engineering with KTH Royal Institute of Technology in Stockholm, Sweden. He received his M.Sc. in Security and Mobile Computing from Aalto University, Finland and KTH, Sweden. His major research interests are mobile computing, distributed systems, and management of large systems.

Gunnar Kreitz has worked at Spotify since 2006, working with, among other things, protocol design, cryptography, and security. Concurrently, he worked on his Ph.D. in Computer Science which he received from KTH Royal Institute of Technology in 2011.

Mikael Goldmann received his M.Sc. degree in Computer Science and Engineering at KTH Royal Institute of Technology, Sweden, in 1988, and Ph.D. in Computer Science from KTH in 1992, where he is an Associate Professor. Since 2007 he is a senior software engineer at Spotify.

Rolf Stadler (`www.ee.kth.se/~stadler`) is a professor at KTH Royal Institute of Technology in Stockholm, Sweden. He holds an M.Sc. degree in mathematics and a Ph.D. in computer science from the University of Zurich. Before joining the faculty at KTH in 2001, he held positions at the IBM Zurich Research Laboratory, Columbia University, and ETH Zürich.

Viktoria Fodor received her M.Sc. and Ph.D. degrees in computer engineering from the Budapest University of Technology and Economics in 1992 and 1999, respectively. In 1999 she joined KTH Royal Institute of Technology, where she now acts as associate professor in the Laboratory for Communication Networks. Her current research interests include network performance evaluation, protocol design, sensor and multimedia networking.