Distrib. Comput. (2006) 18(6): 421–434
DOI 10.1007/s00446-005-0150-7

ORIGINAL ARTICLE

Mema Roussopoulos · Mary Baker

Practical load balancing for content requests in peer-to-peer networks

Received: 5 August 2005 / Accepted: 15 September 2005 / Published online: 10 January 2006
© Springer-Verlag 2006

Abstract This paper studies the problem of balancing the demand for content in a peer-to-peer network across heterogeneous peer nodes that hold replicas of the content. Previous decentralized load balancing techniques in distributed systems base their decisions on periodic updates containing information about load or available capacity observed at the serving entities. We show that these techniques do not work well in the peer-to-peer context; either they do not address peer node heterogeneity, or they suffer from significant load oscillations which result in unutilized capacity. We propose a new decentralized algorithm, Max-Cap, based on the maximum inherent capacities of the replica nodes. We show that unlike previous algorithms, it is not tied to the timeliness or frequency of updates, and consequently requires significantly less update overhead. Yet, Max-Cap can handle the heterogeneity of a peer-to-peer environment without suffering from load oscillations.

Keywords Peer-to-peer networks · Load balancing · Content replica selection

1 Introduction

Peer-to-peer networks are becoming popular architectures for content distribution. The basic premise in such networks is that any one of a set of "replica" nodes can provide the requested content, increasing the availability of interesting content without requiring the presence of any particular serving node.

Many peer-to-peer networks push index entries throughout the overlay peer network in response to lookup queries for specific content [5, 40, 43, 44, 47]. These index entries point to the locations of replica nodes where the particular content can be served, and are typically cached for a finite amount of time, after which they are considered stale. Until now, however, there has been little focus on how an individual peer node should choose among the returned index entries to forward client requests. One reason for considering this choice is load balancing.

In this paper we explore the problem of load-balancing the demand for content in a peer-to-peer network. Our primary goal is to fairly balance the uplink bandwidths of serving peers because upload bandwidth is typically the most scarce resource in a peer-to-peer network [45]. Balancing upload bandwidths is challenging for several reasons. First, in a peer-to-peer network there is no centralized dispatcher that performs the load-balancing of requests; each peer node individually makes its own decision on how to assign incoming requests to replicas. Second, nodes do not typically know the identities of all other peer nodes in the network, and therefore they cannot coordinate with those other nodes. Finally, replica nodes in peer-to-peer networks are not necessarily homogeneous. Some replica nodes may be powerful with good connectivity, whereas others may have limited inherent capacity to handle content requests.

Previous load-balancing techniques base their decisions on periodic or continuous updates containing information on *load* or *available capacity*. We refer to this information as load-balancing information (LBI). Previous approaches

- do not take into account the heterogeneity of peer nodes (e.g., [23, 36]), or
- use techniques such as migration or handoff of tasks (e.g., [30]) that require close coordination amongst serving entities that cannot be achieved in a peer-to-peer environment, or
- suffer from significant load oscillations, or "herd behavior" [36], where peer nodes simultaneously forward an unpredictable number of requests to replicas with low reported load or high reported available capacity causing them to become overloaded. This herd behavior defeats the attempt to provide load-balancing.

M. Roussopoulos (✉)
Harvard University, Cambridge, MA
E-mail: mema@eecs.harvard.edu

M. Baker
HP Labs, Palo Alto, CA
E-mail: mgbaker@hp.com

Most of these techniques also depend on the timeliness of LBI updates. The wide-area nature of peer-to-peer networks and the variation in transfer delays among peer nodes makes guaranteeing the timeliness of updates difficult. Peer nodes will experience varying degrees of staleness in the LBI updates they receive depending on their distance from the source of updates. Moreover, maintaining the timeliness of LBI updates is also costly, since all updates must travel across the Internet to reach interested peer nodes. The smaller the inter-update period and the larger the overlay peer network, the greater the network traffic overhead incurred by LBI updates. Therefore, in a peer-to-peer environment, an effective load-balancing algorithm should not be critically dependent on the timeliness of updates.

In this paper we propose a new and practical load- balancing algorithm, Max-Cap, that makes decisions based on the inherent maximum capacities of the replica nodes. We define maximum capacity as the maximum number of content requests per time unit that a replica claims it can handle. Alternative measures such as maximum (allowed) connections can also be used. The maximum capacity is like a contract by which the replica agrees to abide. If the replica cannot sustain its advertised rate, then it may choose to advertise a new maximum capacity to avoid overload. Max-Cap is not tied to the timeliness or frequency of LBI updates, and as a result, when applied in a peer-to-peer environment, outperforms algorithms based on load or available capacity, whose benefits are heavily dependent on the timeliness of the updates.

We show that Max-Cap takes peer node heterogeneity into account unlike algorithms based on load. While algorithms based on available capacity take heterogeneity into account, we show that surprisingly, they can suffer from significant load oscillations in the presence of small fluctuations in the workload, even when the workload request rate is well below (e.g., 60%) the total maximum capacity of the replicas. On the other hand, Max-Cap avoids overloading replicas in such cases and is more resilient to large fluctuations in workload. This is because a key advantage of Max-Cap is that it uses information that is not affected by changes in the workload.

In a peer-to-peer environment the expectation is that the set of participating nodes changes constantly. Since replica arrivals to and departures from the peer network can affect the information carried in LBI updates, we also compare Max-Cap against availability-based algorithms when the set of replicas continuously changes. We show that Max-Cap is less affected by changes in the replica set than the availability-based algorithms.

We evaluate load-based and availability-based algorithms and compare them with Max-Cap. We use the Controlled Update Propagation (CUP) protocol [43] to propagate the LBI updates required by these algorithms. LBI updates are propagated from replica nodes serving particular content down a conceptual tree, similar to an application-level multicast tree. The vertices of this tree are peer nodes receiving requests for that content. The peer nodes use the LBI updates when choosing to which replica to forward a client request. While we study the load-balancing problem within the context of peer-to-peer systems, the results we presents here apply to any distributed system where the goal is to balance demand for content or service replicated across a set of widely dispersed heterogeneous servers.

The rest of this paper is organized as follows. Section 2 describes the system model under study. Section 3 introduces the algorithms compared. Section 4 presents experimental results showing that in a peer-to-peer environment, Max-Cap outperforms the other algorithms and does so with no or much less overhead. Section 5 describes related work, and Sect. 6 concludes the paper.

2 System model

We assume a peer-to-peer overlay network of widely distributed nodes. The peers store and share content with other peers and are heterogeneous in their capacity to serve content. The placement of a file on a particular peer is decided by the owner of the peer, not by a global placement policy. That is, there is no control over where replicas of a particular file are placed. Finally, the set of participating peers in the system is dynamic as peers enter and leave the system continuously and content availability at a peer can last for as little as a few minutes.

3 The algorithms

We evaluate two different algorithms, Inv-Load and Avail-Cap. Each represents a different class of algorithms that has been proposed in the distributed systems literature. We study how these algorithms perform when applied in a peer-to-peer context and compare them with our proposed algorithm, Max-Cap. These three algorithms depend on different LBI being propagated, but their overall goal is the same: to balance the demand for a specific piece of content fairly across the set of replicas providing that content. In particular, the algorithm should avoid overloading some replicas while underloading others, especially when the aggregate capacity of all replicas is sufficient to handle the content request workload. Moreover, the algorithm should prevent individual replicas from oscillating between being overloaded and underloaded.

Oscillation is undesirable for two reasons. First, many applications limit the number of requests a host can have outstanding. This means that when a replica node is overloaded, it will drop any further requests it receives. This forces the requesting client (or user) to resend its request resulting in additional delay. Even for applications that allow requests to be queued while a replica node is overloaded, the queuing delay incurred will increase. Second, and more importantly, in a peer-to-peer network, the issue of fairness is sensitive. The owners of replica nodes are likely not to

want their nodes to be overloaded while other nodes in the network are underloaded. This is particularly true in peer-to-peer networks where the placement of content on a particular node is decided by the owner of the node, not by a global placement policy. [1] An algorithm that can fairly distribute the request workload without causing replicas to oscillate between being overloaded and underloaded is preferable.

We describe each of the algorithms we evaluate in turn:

Inv-Load: Allocation Proportional to Inverse Load. There are many load-balancing algorithms that base the allocation decision on the load observed at and reported by each of the serving entities (see Related Work Sect. 5). The representative load-based algorithm we examine is Inv-Load, based on the algorithm presented by Genova et al. [23]. In a homogeneous environment, this algorithm has been shown to perform as well as or better than other load-based algorithms. In this algorithm, each peer node in the network chooses to forward a request to a replica with probability inversely proportional to the load reported by the replica. This means that the replica with the smallest reported load (as of the last report received) will receive the most requests from the node. Load is defined as the number of request arrivals at the replica per time unit. Other possible load metrics include the number of request connections open at the replica at reporting time [7] or the request queue length at the replica [19].

When applied in a heterogeneous environment such as a peer-to-peer network, Inv-Load fails. This is intuitive because Inv-Load does not distinguish between replicas observing the same load but having different maximum capacities. For completeness only, we verify this intuition in Appendix B.

We have altered Inv-Load to take heterogeneity into account with a weighting scheme based on maximum capacity. We find that the results for this variation are identical to those of Avail-Cap which is the algorithm we consider next. For this reason, we do not consider Inv-Load nor weighted Inv-Load in the remainder of the paper.

Avail-Cap: Allocation Proportional to Available Capacity. In this algorithm, each peer node chooses to forward a request to a replica with probability proportional to the available capacity reported by the replica. Available capacity is the maximum request rate a replica can handle minus the load (actual request rate) experienced at the replica. This algorithm is based on the algorithm proposed by Zhu et al. [52] for load sharing in a cluster of heterogeneous servers. Avail-Cap takes into account heterogeneity because it distinguishes between nodes that experience the same load but have different maximum capacities.

[1] For example, a global placement policy based on a distributed hash table requires that a particular file be stored on the node responsible for the portion of the hash table to which the name of the file hashes. This is in contrast to a scheme where that node simply stores pointers to the locations of nodes that voluntarily store the content. We thus do not focus on replication-based approaches such as those used in content distribution networks (e.g., Codeen [49] and Beehive [38]) because the premise is that proactive replication will be difficult to enforce in a peer-to-peer network of voluntary, non-dedicated peers.

Avail-Cap appears intuitive because it sends more requests to the replicas that are currently more capable of handling requests. Replicas that are overloaded report an available capacity of zero and are excluded from the allocation decision until they once more report a positive available capacity. Unfortunately, this exclusion can cause Avail-Cap to suffer from severe load oscillations (Sect. 4.2) and is the main reason for its instability.

Both Inv-Load and Avail-Cap depend on the load or available capacity last reported by a replica until the next report is available. Since both these metrics are directly affected by changes in the request workload, both algorithms require that replicas periodically update their LBI. (We assume replicas are not synchronized in when they send reports.) Decreasing the period between two consecutive LBI updates increases the freshness of the LBI at a cost of higher overhead, measured in number of updates pushed through the peer-to-peer network. This overhead is exacerbated with increasing network size. In large peer-to-peer networks, there may be several hops over which updates will have to travel, and the time to do so could be on the order of seconds.

Max-Cap: Allocation Proportional to Maximum Capacity. This is the algorithm we propose. In this algorithm, each peer node chooses to forward a request to a replica with probability proportional to the maximum capacity of the replica. The maximum capacity is a contract each replica advertises indicating the maximum number of requests the replica claims to handle per time unit. Unlike load and available capacity, the maximum capacity of a replica is not affected by changes in the request workload. Therefore, Max-Cap does not depend on the timeliness of LBI updates. In fact, replicas only push updates when they choose to advertise a new maximum capacity. This choice depends on extraneous factors that are unrelated to and independent of the workload (see Sect. 4.7). If replicas rarely choose to change contracts, Max-Cap incurs near-zero overhead. We show that this independence of the timeliness and frequency of LBI updates makes Max-Cap practical and elegant for use in peer-to-peer networks and other distributed systems.

4 Experiments

We first describe the experimental setup including the details of the simulator, the network model, and the parameters we use. We then evaluate the performance of Avail-Cap and Max-Cap under a variety of scenarios observed in real-world peer-to-peer systems.

4.1 Experimental setup

We describe experiments that measure the ability of the Avail-Cap and Max-Cap algorithms to balance requests for a specific piece of content fairly across the replicas holding that piece of content. We say "piece of content" because in

many widely-used peer-to-peer networks, files are divided into equal-sized chunks and clients fetch the chunks of a file in parallel to improve download time (e.g. [1, 2, 16]). A movie file may be much larger than a song file which means it will take a larger number of chunk requests to download it. We assume that if a replica is serving more than one file, it will partition its maximum capacity across the files and advertise for each file accordingly. We describe this further in Sect. 4.8.

In each of the experiments, requests for a particular piece of content are posted at nodes throughout a peer-to-peer network for 3000 seconds. A peer node that receives from a local client a request for that content uses the Controlled Update Propagation (CUP) protocol [43] to retrieve a set of index entries pointing to replica nodes in the network that serve the content, as well as LBI about each replica. Using the LBI, the peer node applies a load-balancing algorithm to choose one of the replica nodes. It then points the client at the chosen replica.

CUP is a cache maintenance protocol that delivers updates to cached index entries and scales efficiently to tens of thousands of nodes. Since the particular mechanism by which LBI is delivered to the peer nodes is not essential to our goal of comparing the load-balancing algorithms, we defer discussion of CUP to Appendix A.

We simulate a peer-to-peer network that implements a *Distributed Hash Table* (DHT) of index entries using the Content-Addressable Network (CAN) approach [40]. In a DHT, a virtual coordinate space is evenly divided amongst the participating peer nodes such that each peer is the authority for a particular portion of the coordinate space. Index entries pointing to locations of content items are mapped onto the virtual coordinate space using a uniform hash function such that the peer whose portion an index entry maps to is the peer responsible for that particular index entry. CAN uses a d-dimensional Cartesian coordinate space on a d-torus. We present results here for $d = 2$. We note that varying the dimensions (and thus the topology of the peer-to-peer overlay network) exhibits similar results because client peers requesting content only use the overlay network to retrieve the index entries pointing to serving peers. They then request the content directly from the serving peers and thus, the content is not transferred through the overlay network itself.

We use Narses, a Java-based discrete-event simulator [4, 24], designed for scalability over large numbers of nodes, large amounts of traffic, and long periods of time. Narses offers facilities for a variety of flow-based network models allowing trade-offs between fast runtimes and accuracy. We are interested in fairly balancing the use of upload links of serving peers. In the Internet, typically the bottleneck link between two communicating hosts is at the edge of the network [9]. We therefore use a network model that assumes no bottleneck link exists in the core of the network. This means that the transfers between two end hosts are limited by their first-link connections to the network. For example, a DSL peer downloading content from a peer on a dial-up modem would be limited by the speed of the DSL link and the modem link. In this case, the modem link would most likely be the limiting factor. With this assumption, Narses does not need to simulate intermediate routers in the network and therefore achieves faster runtime (45-fold speedup) and less memory (28% of that used by the Network Simulator ns-2 [3]) while maintaining accuracy to within 8% [24].

The simulation input parameters include: the number of nodes in the overlay peer-to-peer network, the number of replica nodes holding the content of interest, the maximum capacities of the replica nodes, the distribution of content request inter-arrival times, and the LBI update period, which is the amount of time each replica waits before sending the next LBI update for the Avail-Cap algorithm.

We assign maximum capacities to replica nodes by applying results from previous work that measures the upload capabilities of nodes in peer-to-peer networks [45]. This work has found that the upload capabilities of peer nodes can vary by orders of magnitude. For the particular Gnutella networks measured, around 10% of nodes are connected through dial-up modems, 60% are connected through broadband connections such as cable modem or DSL where the upload speed is about ten times that of dial-up modems, and the remaining 30% have high-end connections with upload speed at least 100 times that of dial-up modems. Therefore we assign maximum capacities of 1, 10, and 100 requests per second to nodes with probability of 0.1, 0.6, and 0.3, respectively.

In all the experiments we present in this paper, the number of nodes in the network is 1024 and each node individually decides how to assign incoming content requests to the replica nodes. To stress-test and compare the load-balancing algorithms, we examine workloads that approach or exceed in magnitude the workload that the serving replica nodes are capable of satisfying (i.e., workloads that require 60–500% of the total maximum capacities of the replica nodes). We use Poisson and Pareto request inter-arrival distributions, both of which have been found to hold in peer-to-peer networks [12, 32]. We find that experiments where we vary the number of nodes in the network but keep the same request workloads exhibit similar behavior. It is, instead, the magnitude of the content request workload that highlights the differences among the load-balancing algorithms.

First, we compare Avail-Cap with Max-Cap for Poisson arrivals and show that while Avail-Cap takes replica heterogeneity into account, it can suffer from significant load oscillations caused by even small fluctuations in the workload (Sect. 4.2). Second, we compare Max-Cap with Avail-Cap for bursty Pareto arrivals (Sect. 4.3). Third, we explain why Avail-Cap suffers (Sect. 4.4). Fourth, we compare the effect on the performances of Avail-Cap and Max-Cap when replicas continuously enter and leave the system (Sect. 4.6). Fifth, we consider the effect on Max-Cap when replicas cannot always honor their advertised maximum capacities because of significant extraneous load (Sect. 4.7). Finally, we examine the tradeoffs that arise in handling multiple objects in Max-Cap (Sect. 4.8).

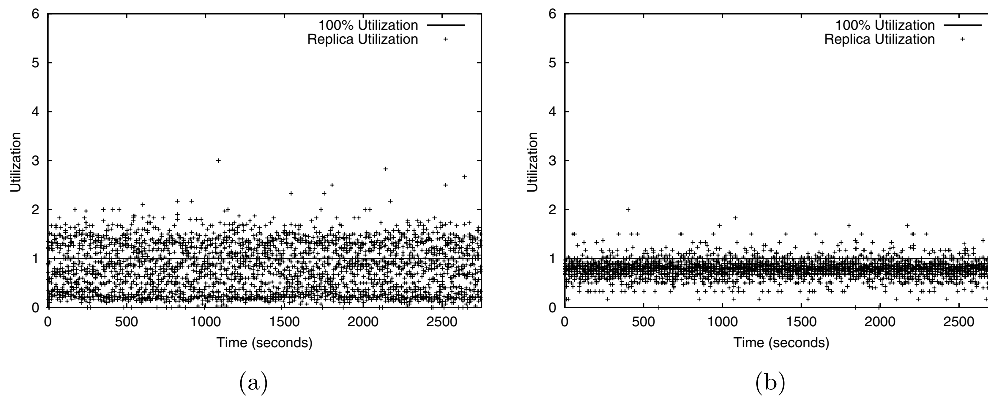


Fig. 1 Replica utilization versus time for **a** Avail-Cap, **b** Max-Cap

4.2 Poisson request arrivals

We first compare Avail-Cap with Max-Cap for an experiment with ten replicas with a Poisson request arrival rate of 80% the total rate that can be handled by the replicas (that is, 80% the total maximum capacity of the replicas). Under such a workload, a good load-balancing algorithm should be able to avoid overloading some replicas while underloading others. For Avail-Cap, we use an inter-update period of one second, which is quite aggressive in a large peer-to-peer network and advantageous for Avail-Cap. For Max-Cap, this parameter is inapplicable since replica nodes do not send periodic updates.

Figure 1 shows a scatterplot of how the utilization of each replica proceeds with time for Avail-Cap and Max-Cap. We define utilization as the request arrival rate observed by a replica divided by the maximum capacity of the replica. In this graph, we do not distinguish among points of different replicas. We see, in Fig. 1a, that Avail-Cap consistently overloads some replicas while underloading others. In contrast, in Fig. 1b, Max-Cap tends to cluster replica utilization at around 80%. We ran this experiment with a range of Poisson arrival rates and found similar results for rates that were 60–100% the total maximum capacity of the replicas. Avail-Cap consistently overloads some replicas while underloading others whereas Max-Cap clusters replica utilization at around X% utilization, where X is the average overall request rate divided by the total maximum capacity of the replicas.

In Avail-Cap, it is not the same replicas that are consistently overloaded or underloaded throughout the experiment. Instead, individual replicas continuously oscillate between being overloaded and severely underloaded. We can see a sampling of this oscillation by looking at the utilizations of some individual replicas over time. In Fig. 2, we plot the utilization over a one minute period in the experiment for a representative replica from each of the replica classes (low, medium, and high maximum capacity). We also plot the ratio of the overall request rate to the total maximum capacity of the replicas and the line $y = 1$ showing 100% utilization. We see that for all replica classes, Avail-Cap suffers from significant oscillation when compared with Max-Cap which causes little or no oscillation above the 100% utilization line. This behavior occurs throughout the experiment.

Figure 3 shows for each replica, the percentage of received requests that arrive while the replica is overloaded for a series of ten experiments, each with ten replicas, for Avail-Cap and Max-Cap respectively. On the x-axis we order replicas according to maximum capacity, with the low-capacity replicas plotted first (replica IDs 1 through 10), followed by the medium-capacity replicas (replica IDs 11-70), followed by the high-capacity replicas (replica IDs 71-100). Avail-Cap with an inter-update period of one second (Fig. 3a) results in much higher percentages than Max-Cap (Fig. 3c). Avail-Cap also causes fairly even percentages at around 40%. This is consistent with the oscillations observed in Fig. 2 where each replica is overloaded for roughly the same amount of time regardless of whether it is a low, medium or high-capacity replica. Moreover, the performance of Avail-Cap is highly dependent on the inter-update period used. As we increase the period and available capacity updates grow more stale, the performance of Avail-Cap suffers more. As an example, in Fig. 3b, we show Avail-Cap with an inter-update period of ten seconds. The overloaded percentages jump up to about 80% across the replicas.

Max-Cap (Fig. 3c) exhibits a step-like behavior with the low-capacity replicas having the highest overloaded percentages, followed by the medium capacity replicas, and then the high-capacity replicas which are never overloaded. This step behavior occurs because the lower-capacity replicas have less tolerance for noise in the random coin tosses the peer nodes perform while assigning requests. They also have less tolerance for small fluctuations in the request rate. As a result, lower-capacity replicas are overloaded more easily than higher-capacity replicas. We also see this in Fig. 2 where for Max-Cap, replicas with lower maximum capacity are overloaded for more time than replicas with higher maximum capacity.

In a peer-to-peer environment, we believe that Max-Cap is a more practical choice than Avail-Cap. First, Max-Cap typically incurs no overhead. Second, Max-Cap can better handle request rates that are below 100% the total maximum

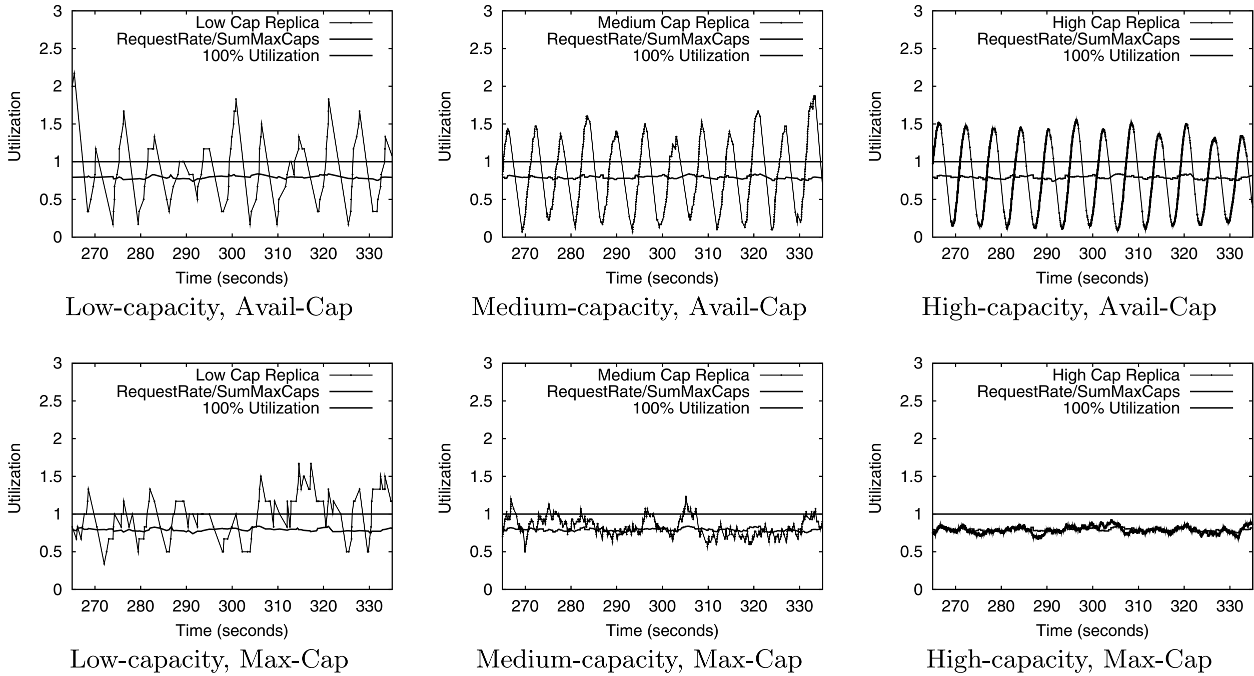


Fig. 2 Replica utilization versus time, for representative low, medium, and high capacity replicas. Top graphs show Avail-Cap, bottom show Max-Cap

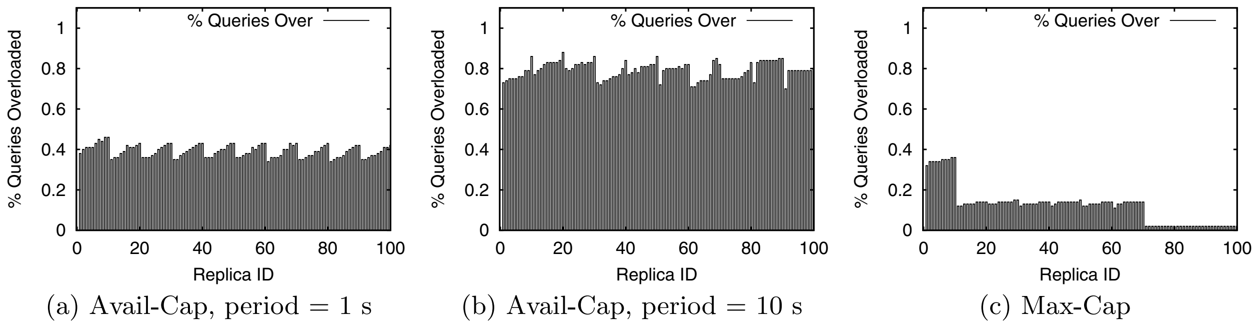


Fig. 3 Percentage Overloaded Requests versus Replica ID, Ten experiments. We show Avail-Cap with an inter-update period of 1 and 10 seconds, and Max-Cap which has no inter-update period

capacity of the replicas and can handle small fluctuations in the workload as are typical in Poisson arrivals.

We next compare how Avail-Cap and Max-Cap perform when workload rates fluctuate beyond the total maximum capacity of the replicas. Such a scenario can occur when requests are bursty, as when inter-request arrival times follow a Pareto distribution.

4.3 Pareto request arrivals

Recent work has observed that in some peer-to-peer networks, request inter-arrivals exhibit burstiness on several time scales [32], making the Pareto distribution a good candidate for modeling these inter-arrival times. Pareto request arrivals are characterized by frequent and intense bursts of requests followed by idle periods of varying lengths [37]. During the bursts, the average request arrival rate can be many times the total maximum capacity of the replicas. We present a representative experiment in which the Pareto shape parameter α and scale parameter κ are 1.1 and 0.0003 respectively. These particular settings cause bursts of up to 230% the total maximum capacity of the replicas. With such intense bursts, no load-balancing algorithm can be expected to keep replicas underloaded. Instead, the best an algorithm can do is to avoid underloading some of the replicas and leaving unutilized capacity that goes wasted.

In Fig. 4, we plot the same representative replica utilizations over a one minute period in the experiment. We also plot the ratio of the overall request rate to the total maximum capacity as well as the $y = 100\%$ utilization line. From the figure we see that Avail-Cap suffers much higher peaks (above the range shown on the y-axis) and lower valleys in replica utilization than Max-Cap. Even when the overall request rate is above 100% of the total maximum capacity,

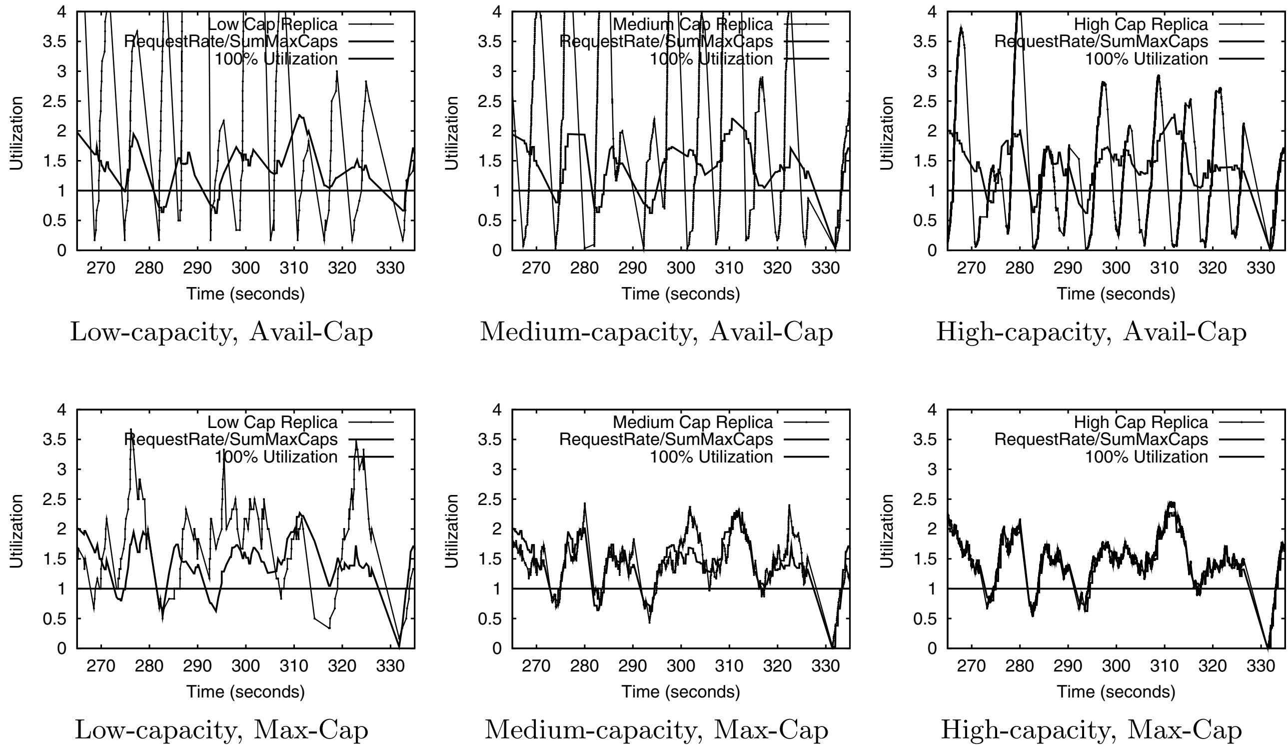


Fig. 4 Representative replica utilization versus time, pareto arrivals. Top graphs show Avail-Cap, bottom show Max-Cap

there are times when the replicas in Avail-Cap are underloaded. In contrast, Max-Cap generally avoids having unutilized capacity when the overall request rate is above 100%. Max-Cap never underutilizes the medium and high capacity replicas and causes little under-utilization of the low capacity replica. This is evident from the fact that the curve for the replica utilization tends to match the curve for the overall request rate in the medium and high capacity graphs.

4.4 Why Avail-Cap can suffer

Avail-Cap can suffer because a cycle is created where the available capacity update of one replica affects a subsequent update of another replica. This affects later allocation decisions made by nodes which in turn affect later replica updates. Consider what happens when a replica is overloaded and reports an available capacity of zero. The report eventually reaches all nodes, causing them to stop directing requests to the replica. The exclusion of the overloaded replica from the allocation decision shifts the burden of the workload to the other replicas. This can cause other replicas to overload and report zero available capacity while the excluded replica experiences a sharp decrease in its utilization. This sharp decrease causes the replica to begin reporting positive available capacity which attracts requests again. Since in the meantime other replicas have become overloaded and excluded from the allocation decision, the replica receives a flock of requests which cause it to become overloaded again. A replica can, therefore, experience severe oscillation where its utilization continuously rises above its maximum capacity and then falls sharply.

We have studied the effect of adding damping to the Avail-Cap algorithm, where a replica advertises an available capacity that is a function of the previously advertised capacity and the current available capacity, or where a replica approaching overload advertises less available capacity than it actually has to ward off future requests and prevent overload from happening. While damping can be beneficial when a single replica is serving content, when multiple replicas are simultaneously damping, we find that the interdependence between updates still results in oscillation.

In Max-Cap, if a replica becomes overloaded, the overload condition is confined to that replica. The same is true in the case of underloaded replicas. Since the overload/underload situations of the replicas are not reported, they do not influence subsequent requests allocated to other replicas. It is this key property that allows Max-Cap to avoid herd behavior.

There are situations where Avail-Cap performs reasonably without suffering from oscillation. We next describe the factors that affect the performance of Avail-Cap to get a clearer picture of when the reactive nature of Avail-Cap is beneficial (or at least not harmful) and when it causes oscillation.

4.5 Factors affecting Avail-Cap

There are four factors that affect the performance of Avail-Cap: the inter-update period U, the inter-request period R, the amount of time T it takes for all nodes in the network to receive the latest update from a replica, and the ratio of

the overall request rate to the total maximum capacity of the replicas. We examine these factors by considering three cases:

Case 1: U is much smaller than R ($U << R$), and T is sufficiently small so that when a replica pushes an update, all peer nodes receive the update before the next request arrival in the network. In this case, Avail-Cap performs well since all nodes have the latest load-balancing information whenever they receive a request.

Case 2: U is long relative to R ($U > R$) and the overall request rate is less than about 60% the total maximum capacity of the replicas.[2] In this case, when a replica overloads, the remaining replicas are able to cover the proportion of requests intended for the overloaded replica because there is a lot of extra capacity in the system. As a result, Avail-Cap avoids oscillations. We see experimental evidence for this in Sect. 4.6.

The 60% threshold is specific to the particular configuration of replicas for which we present results. Other configurations have different threshold percentages but the overall message is the same: the overall request rate must be well below the total maximum capacity of the replicas to avoid oscillation. Over-provisioning to have enough extra capacity in the system so that Avail-Cap can avoid oscillation in this particular case seems a high price to pay for load stability.

Case 3: U is long relative to R ($U > R$) and the overall request rate is more than about 60% the total maximum capacity of the replicas. In this case, as we observe in the experiments above, Avail-Cap can suffer from oscillation. This is because every request that arrives directly affects the available capacity of one of the replicas. Since the request rate is greater than the update rate, an update becomes stale shortly after a replica has pushed it out. However, the replica does not inform the nodes of its changing available capacity until the end of its current update period. By that point many requests have arrived and have been assigned using the previous, stale available capacity information.

In Case 3, Avail-Cap can suffer even if $T = 0$ and updates were to arrive at all nodes instantly after being issued. This is because all nodes would simultaneously avoid an overloaded replica when making the allocation decision until the next update is issued. As T increases, the staleness of the report only exacerbates the performance of Avail-Cap.

In a large peer-to-peer network we expect that T will be on the order of seconds since current peer-to-peer networks with more than 1000 nodes have diameters ranging from a handful to several hops [41]. We consider $U = 1$ second to be as small and aggressive an inter-update period as is practical in a peer-to-peer network. In fact, even one second may be too aggressive due to the overhead it generates. This means that when particular content experiences high popularity, we expect that typically $U + T >> R$. Under such circumstances Avail-Cap is not a good load-balancing choice. For less popular content, where $U + T < R$, Avail-Cap is a feasible choice, although it is unclear whether load-balancing across the replicas is as important here, since the request rate is low.

The performance of Max-Cap is independent of the values of U, R, and T. More importantly, Max-Cap does not require continuous updates; replicas issue updates only if they choose to re-issue new contracts to report changes in their advertised maximum capacities. Therefore, we believe that Max-Cap is a more practical choice in a peer-to-peer context than Avail-Cap.

[2] The 60% threshold is specific to the particular configuration of replicas we present in this paper: 10% low-capacity, 60% medium-capacity, and 30% high-capacity.

4.6 Dynamic replica set

A key characteristic of peer-to-peer networks is that they are subject to constant change; peer nodes continuously enter and leave the system. In this section, we compare Max-Cap with Avail-Cap in two dynamic experiments with a Poisson request arrival rate that is 80% the total maximum capacity of the replicas. The network starts with ten replicas and after a period of 600 seconds, movement into and out of the network begins. In the first experiment, one replica leaves and one replica enters the network every 60 seconds. In the second and much more dynamic experiment, five replicas leave and five replicas enter the network every 60 seconds. Studies have found that in peer-to-peer file sharing systems, the median user session duration of a peer is approximately sixty minutes [45]. However, content may become available on a peer or be deleted from the peer at any point during that user session. This results in content availability that is on the order of a few minutes [15]. We thus show results of experiments that stress-test load-balancing algorithms under this kind of dynamism.

The replicas that leave are randomly chosen. The replicas that enter the network enter with maximum capacities of 1, 10, and 100 with probability of 0.10, 0.60, and 0.30 respectively as in the initial configuration. This means that the total maximum capacity of the active replicas in the network varies throughout the experiment, depending on the differences in capacities of the entering and leaving replicas.

Figures 5a and b show for the first dynamic experiment the utilization of active replicas versus time of Avail-Cap and Max-Cap. Note that points with zero utilization indicate newly entered replicas. The jagged line plots the ratio of the current sum of maximum capacities in the network, S_{curr}, to the original sum of maximum capacities, S_{orig}. With each change in the replica set, the replica utilizations for both Avail-Cap and Max-Cap change. Replica utilizations rise when S_{curr} falls and vice versa. We see that between times 1000 and 1820, S_{curr} is between 1.75 and 2 times S_{orig}, and is more than double the overall workload request rate. During this time period, Avail-Cap performs well because the workload is not demanding and there is plenty of extra capacity in the system (Case 2 in the previous section). However, when at time 1940 S_{curr} falls back to S_{orig}, we see that both algorithms exhibit the same behavior as they do at the start. Max-Cap adjusts nicely and clusters replica utilization at around 80%, while Avail-Cap starts to suffer again.

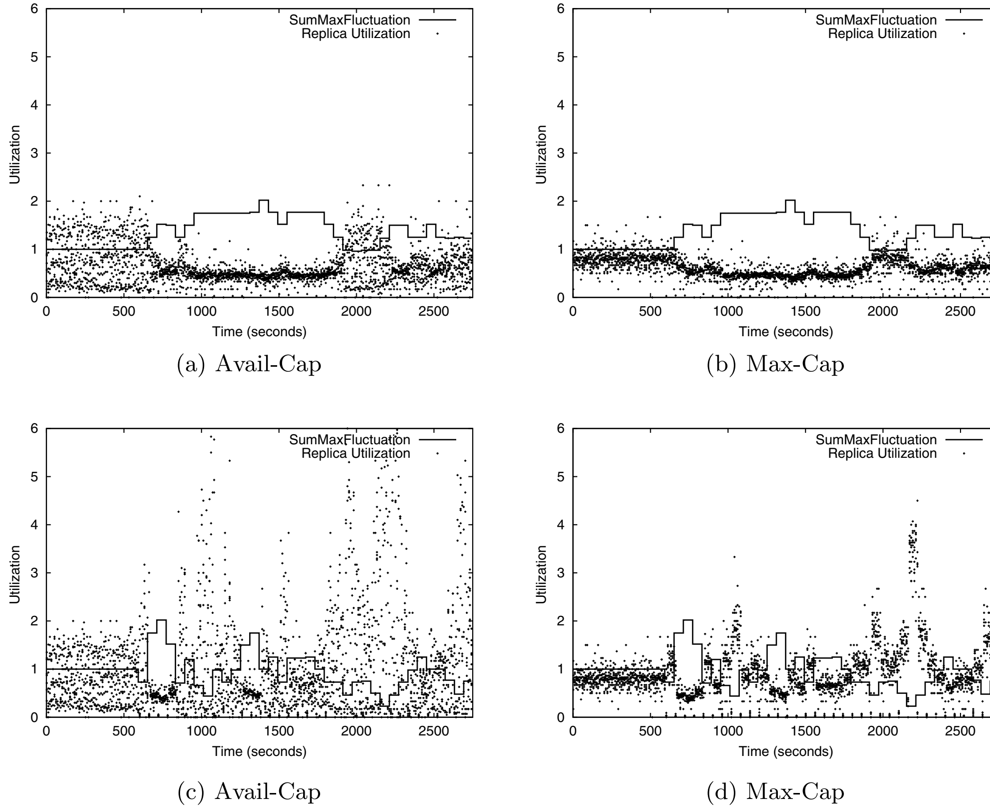


Fig. 5 Replica utilization versus time. Top graphs show one switch every 60 seconds, bottom show 5 switches every 60 seconds

For each data point, we measure the difference between the actual replica utilization and the optimal replica utilization at that time, defined as the overall query rate divided by the total maximum capacity of the replicas. The average utilization difference exhibited by Max-Cap is 4.61 with a standard deviation of 5.77. The average utilization difference for Avail-Cap is 18.29 with a standard deviation of 20.79.

Figures 5c and d show the utilization scatterplot for the second dynamic experiment. We see that changing half the replicas every 60 seconds can dramatically affect S_{curr}. For example, when S_{curr} drops to $0.2S_{\mathrm{orig}}$ at time 2161, the utilizations rise dramatically for both Avail-Cap and Max-Cap. This is because during this period the workload request rate is four times that of S_{curr}. However, as S_{curr} starts to increase, we see that Max-Cap recuperates and adjusts its replica utilization more quickly. The average utilization difference for Max-Cap here is 8.20 with a standard deviation of 14.64 and the average utilization for Avail-Cap is 49.93 with a standard deviation of 69.93.

These dynamic experiments show two things; first, when the workload is not demanding and there is plenty of extra capacity, the behavior of Avail-Cap comes close to that of Max-Cap. However, Avail-Cap suffers more as overall capacity decreases. Second, the reactive nature of Avail-Cap causes it to be affected more by short-lived decreases in the total maximum capacity than Max-Cap is. We conclude that in a dynamic environment such as a peer-to-peer network, Max-Cap continues to be the better choice.

4.7 Extraneous load

As we have shown above, when replicas can honor their maximum capacities, Max-Cap avoids the oscillation that Avail-Cap can suffer, and does so with no update overhead. Occasionally, some replicas may not be able to honor their maximum capacities because of *extraneous load* caused by other applications running on the replicas or network conditions unrelated to the content request workload.

To deal with the possibility of extraneous load, we modify the Max-Cap algorithm slightly to work with honored maximum capacity, which is maximum capacity minus the extraneous load a replica is experiencing. A peer node chooses a replica to forward a content request to with probability proportional to the honored maximum capacity.

As before, we view the honored maximum capacity reported by a replica A as a contract. If A cannot adhere to its contract or has extra capacity to give, and does not report the deficit or surplus, then A alone will be affected and may be overloaded or underloaded since it will be receiving a request share that is proportional to its previous advertised honored maximum capacity. If, on the other hand, replica A chooses to issue a new contract with the new honored maximum capacity, then this can cause a portion of A's workload to shift to the other replicas. This shift however does not affect the contracts of the other replicas. The contract of another replica B is only affected by the extraneous load experienced by B. That is, B may, at some point, choose

to re-issue its contract, but it will do so because it chooses to devote more capacity to its extraneous load, **not** because it has received more requests as a result of A's change in contract. In contrast, in Avail-Cap, the available capacity reported by one replica directly affects the available capacities reported by the others.

In experiments where we inject extraneous load into the replicas, we find that the performances of Max-Cap and Avail-Cap are similar to those seen in the dynamic replicas experiments. This is because when a replica advertises a new honored maximum capacity, it behaves as if that replica were leaving and being replaced by a new replica with a different maximum capacity.

4.8 Handling multiple objects

We have shown that Max-Cap is more practical and more fair than Avail-Cap in allocating load across the replicas serving a particular object. If a peer is serving multiple files, then the peer must partition its maximum capacity across the files and advertise for each file accordingly. The policy used to determine this partition can be independent of the policies at other nodes and might for example, be based on individual file popularity observed at the node. A couple of interesting questions and scenarios arise on how to handle multiple objects:

Should a peer that hosts a very popular object be relieved of its duties for other objects it hosts? As an example, consider the scenario where peer A has high capacity and hosts unpopular object O_1 and highly popular object O_2 and peer B has low capacity and only hosts unpopular O_1. If O_2 receives enough requests to swamp peer A, then should the system send all requests for O_1 to B?

This scenario highlights an important tradeoff. In a peer-to-peer system, there is no control over which objects are stored and served by each peer. Each peer chooses the objects it will store and serve independently of what objects other peers choose. A peer serving a particular object cares that peers serving that same object are getting their fair share of the load with respect to that object. That is, if the main objective is fairness of utilization across serving peers with respect to object O_1, the correct thing would not be to send all requests for O_1 to B, but to continue sending requests to both. Peer A would reject requests it cannot handle due to its workload for O_2 and those requests would eventually be sent to B by the coin-toss retries at the rejected clients. On the other hand, if we assume the main objective is fairness of utilization across serving peers across all objects, then one might argue that the fair thing is to send all requests for O_1 to B. To support this, one would need to modify Max-Cap to have peers take into account what other peers are storing. This unfortunately requires close coordination and the kind of dynamic update propagation from which Avail-Cap suffers under workload fluctuation.

If object popularity changes quickly, how should a peer re-partition capacities across objects and how should a peer advertise this reallocation? As an example, consider the scenario where you have two objects, O_1 and O_2. O_1 is replicated on peers A, B, C. O_2 is replicated on peers C, D, and E. Suppose the maximum capacities of A, B, and C are in the ratio of 1:2:3 and C, D, E are in the ratio of 3:2:1. O_1 starts out popular. After some time, O_2 suddenly becomes popular. Since C is the highest capacity replica serving O_2, it will receive the largest proportion of the requests and experience overload. What should C do?

In Max-Cap, each node individually decides how to partition its maximum capacity across the objects it serves. For example, node C could choose to simply allocate its maximum capacity evenly across the objects it serves. In the example scenario above, node C may allocate 50% of its total capacity to O_1 and 50% to O_2, when O_2's requests begin. This means that the ratio of maximum capacities for nodes A, B, and C will now be 1:2:1.5 for O_1 and 1.5:2:1 for O_2. Another option would be for C to allocate capacity to O_1 and O_2 using an "overbooking" approach, e.g., 70% and 60%. In the latter case, underload (i.e., unused capacity) of O_1 can cover potential overload of O_2 and vice versa. A third option is for C to take advantage of a popularity-tracking algorithm to allocate a capacity to each object that is proportional to the object's popularity. The choice of policy belongs to each individual node and does not require consultation with other nodes in the system.

The choice of whether the peer advertises a re-allocation of capacities is similar to the choice it faces when experiencing extraneous load (Sect. 4.7). If a peer does not advertise the change in its allocations and chooses simply to overbook or borrow unused allocated capacity from one object to serve requests for another, it alone is affected. Any overload the peer experiences as a result is confined to that peer and does not affect other nodes. If the peer chooses to advertise a new maximum capacity as a result of adjusting its allocations, then the effect is similar to the results of the dynamic replica experiments because the node behaves as if it is leaving and being replaced by a new node with a different maximum capacity.

5 Related work

Load-balancing has been the focus of many studies described in the distributed systems literature. We describe here previous techniques that could be applied in a peer-to-peer context. Other techniques that cannot be directly applied in a peer-to-peer context such as task handoff through redirection (e.g., [6, 7, 13]) or process migration (e.g., [30]) from heavily-loaded to lightly-loaded servers in a cluster are described in thesis format [42].

5.1 Load-based algorithms

Of the algorithms based on load, a common approach to performing load-balancing across a set of servers is to choose the server with the least reported load from among a set

of servers. This approach has been used in commercial settings [20] as well as in the research community and performs well in a homogeneous system where the task allocation is performed using complete up-to-date load information [50, 51]. In a system where multiple dispatchers are independently performing the allocation of tasks, this approach however has been shown to behave badly, especially if load information used is stale [21, 22, 33, 36, 46].

Many studies have focused on the strategy of using a subset of the load information available. This has the advantage of incurring less overall traffic because only load updates from some of the servers are sent across the network for an allocation decision. The approach involves first randomly choosing a small number, k, of homogeneous servers and then choosing the server with the smallest reported load from within that set [8, 10, 11, 21, 23, 28, 35, 48]. In particular, for homogeneous systems, Mitzenmacher [35] studies the tradeoffs of various choices of k and various degrees of staleness of load information reported. As the degree of staleness increases, smaller values of k are preferable. In our work, we focus on heterogeneous servers. Max-Cap's probabilistic approach based on maximum capacities allows us to avoid transmitting and using stale information altogether.

Dahlin [19] proposes *load interpretation* algorithms which take into account the age (staleness) of the load information reported by each of a set of distributed homogeneous servers as well as an estimate of the rate at which new requests arrive at the whole system to determine to which server to allocate a request. As Dahlin shows, the primary drawback of these algorithms is that they exhibit poor performance if the estimate of the rate of request arrivals in the system is inaccurate or is not known, as is the case when arrivals are bursty or the workload is unpredictable. Also, these algorithms assume that the clocks of the participating entities are synchronized, which is difficult to achieve in a large peer-to-peer system spread across the Internet. For these reasons, we have focused on developing an algorithm that does not depend on the arrival rate being accurately estimated.

CFS [18] uses proportional allocation of resources to provide load-balancing of files across nodes. CFS is a file system built on top of Chord [47], a peer-to-peer system that implements a distributed hash table. In Chord, peers are responsible for a portion of an identifier space (conceptually depicted as a ring). Queries in search of particular identifiers are routed around the ring towards the peers responsible for those identifiers. CFS deals with heterogeneity in file serving peers by allocating the ring across virtual peers instead of actual peers and allowing administrators to assign to a peer a number of virtual peers that is roughly proportional to the peer's network and storage capabilities.

The CFS authors suggest the possibility of having servers delete virtual peers under high load. This deletion causes other peers to acquire additional virtual peers to cover the identifier space of the deleted virtual peers. Under high overall system load, the authors suggest that a cascade of deletions of virtual peers across the network might occur. Deleting virtual peers under high load is analogous to sending updates reporting reduced available capacity. As we confirm in this paper, as the overall load approaches the capacity of the system, this kind of dynamic and inter-dependent behavior amongst peers can lead to oscillation and is not worth the cost of propagating the updates.

Subsequent studies [25, 26, 39] examine how to prevent this thrashing problem by moving virtual servers from heavily-loaded to lightly-load peers only if the transfer does not cause the load on the light peers to surpass some pre-defined threshold. This and other studies on structured Distributed Hash Tables [27, 29, 31] examine the load-balancing problem from the perspective that each item is stored on one virtual server and all requests for that item go to the peer that owns that virtual server. In our study, we have several replica nodes that can serve the item and the problem is how to enable large numbers of widely-dispersed client nodes to independently choose from amongst the replica nodes such that the overall demand is fairly distributed across those replicas.

5.2 Available-capacity-based algorithms

Of the algorithms based on available capacity, one common approach has been to choose amongst a set of servers based on the available capacity of each server [52] or the available bandwidth in the network to each server [14]. The server with the highest available capacity/bandwidth is chosen by a client with a request. The assumption here is that the reported available capacity/bandwidth will continue to be valid until the chosen server has finished servicing the client's request.

Another approach is to exclude servers that fail some utilization threshold and to choose from the remaining servers. Mirchandaney et al. [34] and Shivaratri et al. [46] classify machines as lightly-utilized or heavily-utilized and then choose randomly from the lightly-utilized servers. This work focuses on local-area distributed systems. Colajanni et al. use this approach to enhance round-robin DNS load-balancing across a set of widely distributed heterogeneous web servers [17]. The maximum capacities of the most capable servers are at most a factor of three that of the least capable servers. As we see in Sect. 4.2, when applied in the context of a peer-to-peer network where the maximum capacities of the replicas can differ by orders of magnitude, excluding a serving node temporarily from the allocation decision can result in severe load oscillation.

6 Conclusions

In this paper we examine the problem of load-balancing in a peer-to-peer network where the goal is to distribute the demand for a particular content fairly across the set of replica nodes that serve that content. Existing load-balancing algorithms proposed in the distributed systems literature are not

appropriate for a peer-to-peer network. Algorithms based purely on load do not handle peer heterogeneity. Algorithms based on available capacity can suffer from load oscillations even when the workload request rate is well below (e.g., 60%) the total maximum capacity of the replicas. These load oscillations result in unutilized capacity. We provide a model that shows when available capacity information detracts rather than enhances the allocation decision.

We propose and evaluate Max-Cap, a practical load-balancing algorithm that handles heterogeneity, yet does not suffer from oscillations even as the workload rate approaches the total maximum capacity of the replicas. It adjusts better to large fluctuations in the workload and constantly changing replica sets. Moreover, Max-Cap incurs much less overhead, since unlike algorithms based on available capacity, it avoids chasing dynamic updates that are interdependent. The contract issued by one replica is independent of the contracts issued by others. In fact, if replicas rarely choose to change contracts, Max-Cap incurs near-zero overhead regardless of the network size. We believe this makes Max-Cap a practical and elegant algorithm for load-balancing in peer-to-peer networks and in other distributed systems with a large number of client dispatchers and heterogeneous servers that are widely dispersed across the Internet.

Acknowledgements The work presented here has benefited greatly from discussions with Petros Maniatis, Armando Fox, Nick McKeown, and Rajeev Motwani. We thank them for their invaluable feedback. We also would like to thank the anonymous reviewers who suggested ways to improve this paper, including suggesting the multiple-object scenarios described above. This research has been supported by the National Science Foundation (Grant No. 0446522), the Stanford Networking Research Center, and by DARPA (contract N66001-00-C-8015).

Appendix A: Propagating LBI Updates

In this appendix we briefly describe how we leverage the CUP protocol [43] to study the load-balancing problem in a peer-to-peer context. CUP is a protocol for maintaining caches of index entries in peer-to-peer networks. We summarize how CUP works over structured peer-to-peer networks. In such networks, lookup queries for particular content follow a well-defined path from the querying node toward an *authority node*, which is guaranteed to know the location of the content within the network [40, 44, 47].

In CUP every node in the peer-to-peer network maintains two logical channels per neighbor: a query channel and an update channel. The query channel is used to forward lookup queries for content of interest to the neighbor that is closest to the authority node for that content. The update channel is used to forward query responses asynchronously to a neighbor. These query responses contain sets of index entries that point to nodes holding the content in question. The update channel is also used to update the index entries that are cached at the neighbor.

Figure 6 shows a snapshot of CUP in progress in a network of seven nodes. The four logical channels are shown between each pair of nodes. The left half of each node shows the set of content items for which the node is the authority. The right half shows the set of content items for which the node has cached index entries as a result of handling lookup queries. For example, node A is the authority node for content $K3$ and nodes C,D,E,F, and G have cached index entries for content $K3$. The process of querying and updating index entries for a particular content K forms a CUP tree whose root is the authority node

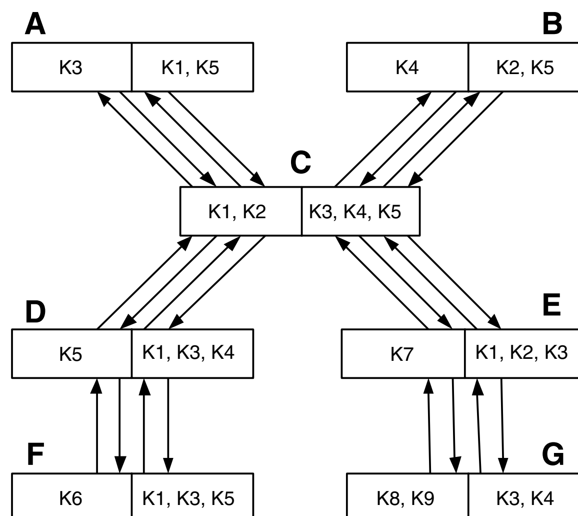


Fig. 6 CUP trees

for content K. The branches of the tree are formed by the paths traveled by lookup queries from other nodes in the network. For example, in Fig. 6, node A is the root of the CUP tree for $K3$ and branch {F,D,C,A} has grown as a result of a lookup query for $K3$ at node F.

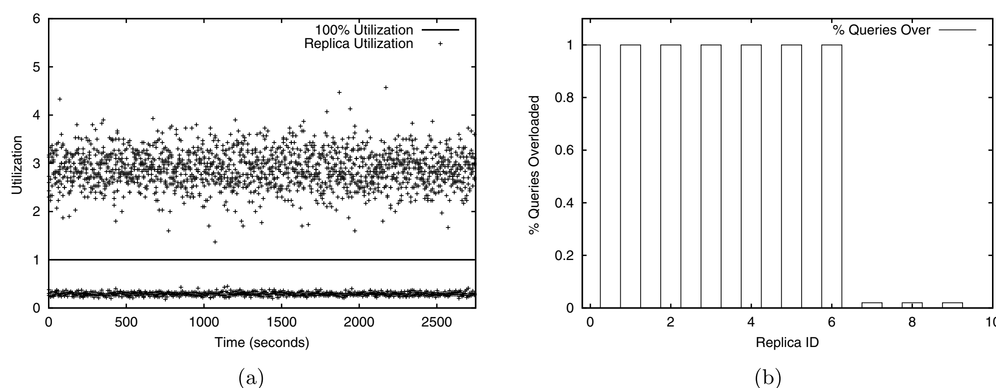
It is the authority node A for content $K3$ which is guaranteed to know the location of all nodes, called *content replica nodes*, that serve content $K3$. Replica nodes first send birth messages to authority A to indicate they are serving content $K3$. They may also send periodic refreshes or invalidation messages to A to indicate they are still serving or no longer serving the content. A then forwards on any birth, refresh or invalidation messages it receives, which are propagated down the CUP tree to all interested nodes in the network. For example, in Fig. 6 any update messages for index entries associated with content $K3$ that arrive at A from replica nodes are forwarded down the $K3$ CUP tree to C at level 1, D and E at level 2, and F and G at level 3.

CUP reduces the average latency of content search queries by as much as an order of magnitude across a variety of workloads and scales to tens of thousands of nodes. For more details, we refer the reader to prior published work [43].

We leverage CUP to propagate updates to load balancing information such as replica load or available capacity to interested peer nodes throughout the overlay network. These peer nodes use this information when choosing to which replica a client request should be forwarded.

Appendix B: Inv-Load and Heterogeneity

In this appendix, we examine the performance of Inv-Load in a heterogeneous peer-to-peer environment. We use a fairly short inter-update period of one second, which is quite aggressive in a large peer-to-peer network. We have ten replica nodes that serve the content item of interest. We generate request rates for that item according to a Poisson process with arrival rate that is 80% the total maximum capacity of the replicas. Under such a workload, a good load-balancing algorithm should be able to avoid overloading some replicas while underloading others. Figure 7a shows a scatterplot of how the utilization of each replica proceeds with time when using Inv-Load. We define utilization as the request arrival rate (load) observed by a replica divided by the maximum request rate the replica can handle (maximum capacity) of the replica. As described in Sect. 1, we assign maximum capacities of 1, 10, and 100 requests per second to nodes with probability of 0.1, 0.6, and 0.3, respectively. In this graph, we do not distinguish among points of different replicas. We see that throughout the simulation, at



(a) (b)

Fig. 7 Inv-Load: **a** Replica Utilization versus Time, **b** Percentage Overloaded Requests versus Replica ID

any point in time, some replicas are severely overutilized (over 250%) while others are lightly underutilized (around 25%).

Figure 7b shows for each replica, the percentage of all received requests that arrive while the replica is overloaded. The replicas that receive almost 100% of their requests while overloaded (i.e., replicas 0-6) are the low and medium-capacity replicas. The replicas that receive almost no requests while overloaded (i.e., replicas 7-9) are the high-capacity replicas. We see that Inv-Load penalizes the less capable replicas while giving the high-capacity replicas an easy time.

Inv-Load is designed to perform well in a homogeneous environment. When applied in a heterogeneous environment such as a peer-to-peer network, it fails. As we see in Sect. 4.2, Max-Cap is much better suited for heterogeneous environments. Moreover, a nice bonus is that Max-Cap has better load-balancing performance than Inv-Load even in a homogeneous environment since it does not require continuous inter-dependent updates. Since the focus in this paper is on heterogeneous environments, such as those found in peer-to-peer networks, we refer the reader to the first author's thesis for more information about homogeneous environments [42].

References

1. Http://www.overnet.com
2. Http://www.kazaa.com
3. Ns Project. Http://www.isi.edu/nsnam/ns/
4. Project: The Narses Network Simulator. Http://sourceforge.net/projects/narses/
5. The Gnutella Protocol Specification v0.4. Http://gnutella.wego.com
6. Andresen, D., Yang, T., Ibarra, O.H.: Towards a Scalable Distributed WWW Server on Networked Workstations. Journ. of Parallel and Distributed Computing **42**, 91–100 (1996)
7. Aversa, L., Bestavros, A.: Load balancing a cluster of web servers using distributed packet rewriting. In: IEEE Intl Performance, Computing, and Communications Conference (2000)
8. Azar, Y., Broder, A., Karlin, A., Upfal, E.: Balanced allocations. In: ACM Symp. on Theory of Computing (1994)
9. Blake, C., Rodrigues, R.: High availability, scalable storage, dynamic peer networks: pick two. In: HotOS (2003)
10. Byers, J., Considine, J., Mitzenmacher, M.: Simple load balancing for distributed hash tables. In: IPTPS (2003)
11. Byers, J., Considine, J., Mitzenmacher, M.: Geometric generalizations of the powser of two choices. In: SPAA (2004)
12. Cao, P.: Search and Replication in Unstructured Peer-to-Peer Networks (2002). Talk at http://netseminar.stanford.edu/
13. Cardellini, V., Colajanni, M., Yu, P.: Redirection algorithms for load sharing in distributed web server systems. In: ICDCS (1999)
14. Carter, R., Crovella, M.: Server selection using dynamic path characterization in wide-area networks. In: Infocom (1997)
15. Chu, J., Labonte, K., Levine, B.N.: Availability and locality measurements of peer-to-peer file systems. In: Proc. ITCom: Scalability and Traffic Control in IP Networks II Conferences (2002)
16. Cohen, B.: Incentives build robustness in bittorrent. In: Workshop on the Economics of Peer-to-Peer Systems (2003)
17. Colajanni, M., Yu, P.S., Cardellini, V.: Dynamic load balancing in geographically distributed heterogeneous web servers. In: ICDCS (1998)
18. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: SOSP (2001)
19. Dahlin, M.: Interpreting stale load information. In: ICDCS (1999)
20. Delgadillo, K.: Cisco Distributed Director. Cisco Systems Whitepaper (1997)
21. Eager, D., Lazowska, E., Zahorjan, J.: Adaptive load sharing in homogeneous distributed systems. IEEE Trans. on Software Engineering **12**(5), 662–675 (1986)
22. Fox, A., Gribble, S.D., Chawathe, Y., Brewer, E.A., Gauthier, P.: Cluster-based scalable network services. In: Symposium on Operating Systems Principles (1997)
23. Genova, Z., Christensen, K.J.: Challenges in URL switching for implementing globally distributed web sites. In: Workshop on Scalable Web Services (2000)
24. Giuli, T.J., Baker, M.: Narses: A Scalable, flow-based network simulator. Technical Report cs.PF/0211024, Computer Science Department, Stanford University, Stanford, CA, USA (2002)
25. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in dynamic structured P2P systems. In: Infocom (2004)
26. Godfrey, B., Stoica, I.: Heterogeneity and load balance in distributed hash tables. In: Infocom (2005)
27. Karger, D., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: SPAA (2004)
28. Karp, R., Luby, M., Heide, F.M.: Efficient PRAM simulation on a distributed memory machine. In: 24th ACM Symposium on Theory of Computing (1992)
29. Ledlie, J., Seltzer, M.: Distributed, secure load balancing with skew, heterogeneity, and churn. In: Infocom (2005)
30. Lu, C., Lau, S.M.: An adaptive load balancing algorithm for heterogeneous distributed systems with multiple task classes. In: ICDCS (1996)
31. Manku, G.S.: Balanced binary trees for ID management and load balance in distributed hash tables. In: PODC (2004)
32. Markatos, E.P.: Tracing a large-scale peer-to-peer system: an hour in the life of gnutella. In: 2nd IEEE/ACM Intl Symposium on Cluster Computing and the Grid (2002)
33. Mirchandaney, R., Towsley, D., Stankovic, J.: Analysis of the Effects of Delays on Load Sharing. IEEE Trans. on Computers **38**, 1513–1525 (1989)
34. Mirchandaney, R., Towsley, D., Stankovic, J.: Adaptive Load Sharing in Heterogeneous Distributed Systems. Journal of Parallel and Distributed Computing **9**, 331–346 (1990)

35. Mitzenmacher, M.: The Power of Two Choices in Randomized Load Balancing. PhD thesis, UC Berkeley (1996)
36. Mitzenmacher, M.: How useful is old information? In: PODC (1997)
37. Paxson, V., Floyd, S.: Wide-area Traffic: The Failure of Poisson Modeling. IEEE/ACM Transactions on Networking **3**(3), (1995)
38. Ramasubramanian, V., Sirer, E.G.: Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In: Proceedings of Networked System Design and Implementation (NSDI) (2004)
39. Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in structured P2P systems. In: IPTPS (2003)
40. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: SIGCOMM (2001)
41. Ripeanu, M., Foster, I.: Mapping the gnutella network: macroscopic properties of large-scale peer-to-peer systems. In: IPTPS (2002)
42. Roussopoulos, M.: Controlled Update Propagation in Peer-to-Peer Networks. PhD thesis, Stanford University (2002)
43. Roussopoulos, M., Baker, M.: CUP: Controlled update propagation in peer to peer networks. In: USENIX Annual Technical Conference (2003)
44. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: MiddleWare (2001)
45. Saroiu, S., Gummadi, P.K., Gribble, S.D.: A measurement study of peer-to-peer file sharing systems. In: Proc. of Multimedia Computing and Networking (2002)
46. Shivaratri, N., Krueger, P., Singhal, M.: Load Distributing for Locally Distributed Systems. IEEE Computer pp. 33–44 (1992)
47. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM (2001)
48. Vvedenskaya, N., Dobrushin, R., Karpelevich, F.: Queuing Systems with Selection of the Shortest of Two Queues: an Asymptotic Approach. Problems of Information Transmission **32**, 15–27 (1996)
49. Wang, L., Pai, V., Peterson, L.: The effectiveness of request redirection on cdn robustness. In: OSDI (2002)
50. Weber, R.: On the Optimal Assignment of Customers to Parallel Servers. Journal of Applied Probability **15**, 406–413 (1978)
51. Winston, W.: Optimality of the Shortest Line Discipline. Journal of Applied Probability **14**, 181–189 (1977)
52. Zhu, H., Yang, T., Zheng, Q., Watson, D., Ibarra, O.H., Smith, T.: Adaptive load sharing for clustered digital library services. In: 7th IEEE HPDC (1998)

Mema Roussopoulos is an Assistant Professor of Computer Science on the Gordon McKay Endowment at Harvard University. Before joining Harvard, she was a Postdoctoral Fellow in the Computer Science Department at Stanford University. She received her PhD and Master's degrees in Computer Science from Stanford, and her Bachelor's degree in Computer Science from the University of Maryland at College Park. Her interests are in the areas of distributed systems, networking, and mobile and wireless computing.

Mary Baker is a Senior Research Scientist at HP Labs. Her research interests include distributed systems, networks, mobile systems, security, and digital preservation. Before joining HP Labs she was on the faculty of the computer science department at Stanford University where she ran the MosquitoNet project. She received her PhD from the University of California at Berkeley.