Low Latency via Redundancy

Ashish Vulimiri
UIUC
vulimir1@illinois.edu

P. Brighten Godfrey
UIUC
pbg@illinois.edu

Radhika Mittal
UC Berkeley
radhika@eecs.berkeley.edu

Justine Sherry
UC Berkeley
justine@eecs.berkeley.edu

Sylvia Ratnasamy
UC Berkeley
sylvia@eecs.berkeley.edu

Scott Shenker
UC Berkeley and ICSI
shenker@icsi.berkeley.edu

ABSTRACT

Low latency is critical for interactive networked applications. But while we know how to scale systems to increase capacity, reducing latency — especially the tail of the latency distribution — can be much more difficult. In this paper, we argue that the use of redundancy is an effective way to convert extra capacity into reduced latency. By initiating redundant operations across diverse resources and using the first result which completes, redundancy improves a system's latency even under exceptional conditions. We study the tradeoff with added system utilization, characterizing the situations in which replicating all tasks reduces mean latency. We then demonstrate empirically that replicating all operations can result in significant mean and tail latency reduction in real-world systems including DNS queries, database servers, and packet forwarding within networks.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General

Keywords

Latency; Reliability; Performance

1. INTRODUCTION

Low latency is important for humans. Even slightly higher web page load times can significantly reduce visits from users and revenue, as demonstrated by several sites [28]. For example, injecting just 400 milliseconds of artificial delay into Google search results caused the delayed users to perform 0.74% fewer searches after 4-6 weeks [9]. A 500 millisecond delay in the Bing search engine reduced revenue per user by 1.2%, or 4.3% with a 2-second delay [28]. Human-computer interaction studies similarly show that people react to small differences in the delay of operations (see [17] and references therein).

Achieving consistent low latency is challenging. Modern applications are highly distributed, and likely to get more so as cloud computing separates users from their data and computation. Moreover, application-level operations often require tens or hundreds of tasks to complete — due to many objects comprising a single web page [25], or aggregation of many back-end queries to produce a front-end result [2, 14]. This means individual tasks may have latency budgets on the order of a few milliseconds or tens of milliseconds, and *the tail* of the latency distribution is critical. Such outliers are difficult to eliminate because they have many sources in complex systems; even in a well-provisioned system where individual operations usually work, some amount of uncertainty is pervasive. Thus, latency is a difficult challenge for networked systems: How do we make the other side of the world feel like it is *right here*, even under exceptional conditions?

One powerful technique to reduce latency is *redundancy*: Initiate an operation multiple times, using as diverse resources as possible, and use the first result which completes. Consider a host that queries multiple DNS servers in parallel to resolve a name. The overall latency is the minimum of the delays across each query, thus potentially reducing both the mean and the tail of the latency distribution. For example, a replicated DNS query could mask spikes in latency due to a cache miss, network congestion, packet loss, a slow server, and so on. The power of this technique is that it reduces latency precisely under the most challenging conditions—when delays or failures are unpredictable—and it does so without needing any information about what these conditions might be.

Redundancy has been employed to reduce latency in several networked systems: notably, as a way to deal with failures in DTNs [21], in a multi-homed web proxy overlay [5], and in limited cases in distributed job execution frameworks [4, 15, 32].

However, these systems are exceptions rather than the rule. Redundant queries are typically eschewed, whether across the Internet or within data centers. The reason is rather obvious: duplicating every operation doubles system utilization, or increases usage fees for bandwidth and computation. The default assumption in system design is that doing less work is best.

But when exactly is that natural assumption valid? Despite the fact that redundancy is a fundamental technique that has been used in certain systems to reduce latency, the

CoNEXT'13, December 9-12, 2013, Santa Barbara, California, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2101-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535372.2535392.

conditions under which it is effective are not well understood — and we believe as a result, it is not widely used.

In this paper, we argue that redundancy is an effective *general* technique to achieve low latency in networked systems. Our results show that redundancy could be used much more commonly than it is, and in many systems represents a missed opportunity.

Making that argument requires an understanding of when replication improves latency and when it does not. Consider a system with a fixed set of servers, in which queries are relatively inexpensive for clients to send. If a single client duplicates its queries, its latency is likely to decrease, but it also affects other users in the system to some degree. If *all* clients duplicate every query, then every client has the benefit of receiving the faster of two responses (thus decreasing mean latency) but system utilization has doubled (thus increasing mean latency). It is not immediately obvious under what conditions the former or latter effect dominates.

Our first key contribution is to characterize when such global redundancy improves latency. We introduce a queueing model of query replication, giving an analysis of the expected response time as a function of system utilization and server-side service time distribution. Our analysis and extensive simulations demonstrate that assuming the client-side cost of replication is low, there is a server-side *threshold load* below which replication always improves mean latency. We give a crisp conjecture, with substantial evidence, that this threshold *always lies between 25% and 50% utilization regardless of the service time distribution*, and that it can approach 50% arbitrarily closely as variance in service time increases. Our results indicate that redundancy should have a net positive impact in a large class of systems, despite the extra load that it adds.

While our analysis only addresses mean latency, we believe (and our experimental results below will demonstrate) that redundancy improves both the mean and the tail.

Our second key contribution is to demonstrate multiple practical application scenarios in which replication empirically provides substantial benefit, yet is not generally used today. These scenarios, along with scenarios in which replication is *not* effective, corroborate the results of our analysis. More specifically:

- **DNS queries across the wide area.** Querying multiple DNS servers reduces the fraction of responses later than 500 ms by 6.5×, while the fraction later than 1.5 sec is reduced by 50×, compared with a non-replicated query to the *best* individual DNS server. Although this incurs added load on DNS servers, replication saves more than 100 msec per KB of added traffic, so that it is more than an order of magnitude better than an estimated cost-effectiveness threshold [29, 30]. Similarly, a simple analysis indicates that replicating TCP connection establishment packets can save roughly 170 msec (in the mean) and 880 msec (in the tail) per KB of added traffic.

- **Database queries within a data center.** We implement query replication in a database system similar to a web service, where a set of clients continually read objects from a set of back-end servers. Our results indicate that when most queries are served from disk and file sizes are small, replication provides substantial latency reduction of up to 2× in the mean and up to 8× in the tail. As predicted by our analysis, mean latency is reduced up to a server-side *threshold load* of 30-40%. We also show that when retrieved files become large or the database resides in memory, replication does not offer a benefit. This occurs across both a web service database and the memcached in-memory database, and is consistent with our analysis: in both cases (large or in-memory files), the client-side cost of replication becomes significant *relative to* the mean query latency.

- **In-network packet replication.** We design a simple strategy for switches, to replicate the initial packets of a flow but treat them as lower priority. This offers an alternate mechanism to limit the negative effect of increased utilization, and simulations indicate it can yield up to a 38% median end-to-end latency reduction for short flows.

In summary, as system designers we typically build scalable systems by avoiding unnecessary work. The significance of our results is to characterize a large class of cases in which duplicated work is a useful and elegant way to achieve robustness to variable conditions and thus reduce latency.

2. SYSTEM VIEW

In this section we characterize the tradeoff between the benefit (fastest of multiple options) and the cost (doing more work) due to redundancy from the perspective of a system designer optimizing a *fixed set* of resources. We analyze this tradeoff in an abstract queueing model (§2.1) and evaluate it empirically in two applications: a disk-backed database (§2.2) and an in-memory cache (§2.3). We then discuss a setting in which the cost of overhead can be eliminated: a data center network capable of deprioritizing redundant traffic (§2.4).

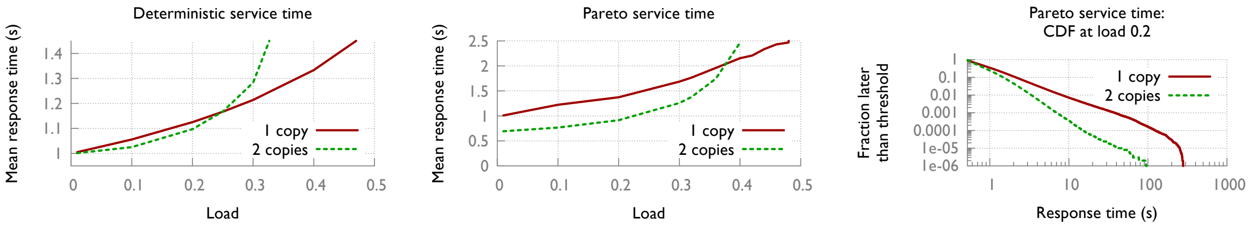
§3 considers the scenario where the available resources are provisioned according to payment, rather than static.

2.1 System view: Queueing analysis

Two factors are at play in a system with redundancy. Replication reduces latency by taking the faster of two (or more) options to complete, but it also worsens latency by increasing the overall utilization. In this section, we study the interaction between these two factors in an abstract queueing model.

We assume a set of N independent, identical servers, each with the same service time distribution S. Requests arrive in the system according to a Poisson process, and k copies are made of each arriving request and enqueued at k of the N servers, chosen uniformly at random. To start with, we will assume that redundancy is "free" for the clients — that it adds no appreciable penalty apart from an increase in server utilization. We consider the effect of client-side overhead later in this section.

Figures 1(a) and 1(b) show results from a simulation of this queueing model, measuring the mean response time (queueing delay + service time) as a function of load with two different service time distributions. Replication improves the mean, but provides the greatest benefit in the tail, for example reducing the 99.9th percentile by 5× under



(a) Mean: deterministic (b) Mean: Pareto (c) CDF: Pareto

Figure 1: A first example of the effect of replication, showing response times when service time distribution is deterministic and Pareto ($\alpha = 2.1$)

Pareto service times. Note the thresholding effect: in both systems, there is a *threshold load* below which redundancy always helps improve mean latency, but beyond which the extra load it adds overwhelms any latency reduction that it achieves. The threshold is higher — i.e., redundancy helps over a larger range of loads — when the service time distribution is more variable.

The threshold load, defined formally as the largest utilization below which replicating every request to 2 servers always helps mean response time, will be our metric of interest in this section. We investigate the effect of the service time distribution on the threshold load both analytically and in simulations of the queueing model. Our results, in brief:

1. If redundancy adds no client-side cost (meaning server-side effects are all that matter), there is strong evidence to suggest that no matter what the service time distribution, the threshold load has to be more than 25%.
2. In general, the higher the variability in the service-time distribution, the larger the performance improvement achieved.
3. Client-side overhead can diminish the performance improvement due to redundancy. In particular, the threshold load can go below 25% if redundancy adds a client-side processing overhead that is significant compared to the server-side service time.

If redundancy adds no client-side cost

Our analytical results rely on a simplifying approximation: we assume that the states of the queues at the servers evolve completely independently of each other, so that the average response time for a replicated query can be computed by taking the average of the minimum of two *independent* samples of the response time distribution at each server. This is not quite accurate because of the correlation introduced by replicated arrivals, but we believe this is a reasonable approximation when the number of servers N is sufficiently large. In a range of service time distributions, we found that the mean response time computed using this approximation was within 3% of the value observed in simulations with $N = 10$, and within 0.1% of the value observed in simulations with $N = 20$.

We start with a simple, analytically-tractable special case: when the service times at each server are exponentially distributed. A closed form expression for the response time CDF exists in this case, and it can be used to establish the following result.

THEOREM 1. *Within the independence approximation, if the service times at every server are i.i.d. exponentially distributed, the threshold load is* 33%.

PROOF. Assume, without loss of generality, that the mean service time at each server is 1 second. Suppose requests arrive at a rate of ρ queries per second per server.

Without replication, each server evolves as an M/M/1 queue with departure rate 1 and arrival rate ρ. The response time of each server is therefore exponentially distributed with rate $1 - \rho$ [6], and the mean response time is $\frac{1}{1-\rho}$.

With replication, each server is an M/M/1 queue with departure rate 1 and arrival rate 2ρ. The response time of each server is exponentially distributed with rate $1 - 2\rho$, but each query now takes the minimum of two independent samples from this distribution, so that the mean response time of each query is $\frac{1}{2(1-2\rho)}$.

Now replication results in a smaller response time if and only if $\frac{1}{2(1-2\rho)} < \frac{1}{1-\rho}$, i.e., when $\rho < \frac{1}{3}$. □

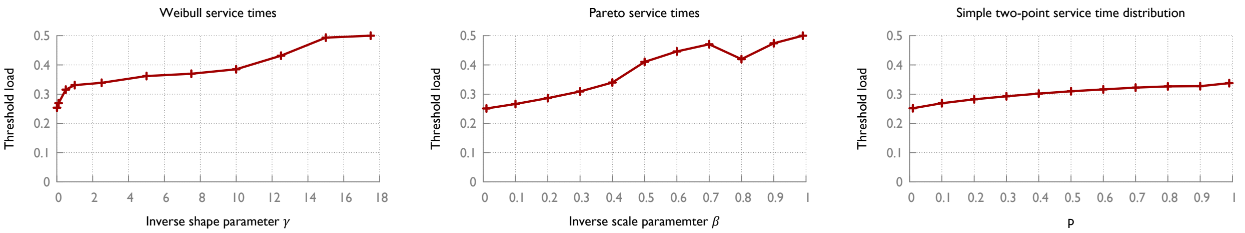
While we focus on the $k = 2$ case in this section, the analysis in this theorem can be easily extended to arbitrary levels of replication k.

Note that in this special case, since the response times are exponentially distributed, the fact that replication improves mean response time automatically implies a stronger distributional dominance result: replication also improves the pth percentile response time for every p. However, in general, an improvement in the mean does not automatically imply stochastic dominance.

In the general service time case, two natural (service-time independent) bounds on the threshold load exist.

First, the threshold load cannot exceed 50% load in any system. This is easy to see: if the base load is above 50%, replication would push total load above 100%. It turns out that this trivial upper bound is tight — there are families of heavy-tailed high-variance service times for which the threshold load goes arbitrarily close to 50%. See Figures 2(a) and 2(b).

Second, we intuitively expect replication to help more as the service time distribution becomes more variable. Figure 2 validates this trend in three different families of distributions. Therefore, it is reasonable to expect that the worst-case for replication is when the service time is completely deterministic. However, even in this case the threshold load is strictly positive because there is still variability in the system due to the stochastic nature of the arrival process. With



(a) Weibull (b) Pareto (c) Two-point discrete distribution

Figure 2: Effect of increasing variance on the threshold load in three families of unit-mean distributions: Pareto, Weibull, and a simple two-point discrete distribution (service time = 0.5 with probability p, $\frac{1-0.5p}{1-p}$ with probability $1-p$). In all three cases the variance is 0 at $x = 0$ and increases along the x-axis, going to infinity at the right edge of the plot.

the Poisson arrivals that we assume, the threshold load with deterministic service time turns out to be slightly less than 26% — more precisely, ≈ 25.82% — based on simulations of the queueing model, as shown in the leftmost point in Figure 2(c).

We conjecture that this is, in fact, a lower bound on the threshold load in an arbitrary system.

Conjecture 1. *Deterministic service time is the worst case for replication: there is no service time distribution in which the threshold load is below the (≈ 26%) threshold when the service time is deterministic.*

The primary difficulty in resolving the conjecture is that general response time distributions are hard to handle analytically, especially since in order to quantify the effect of taking the minimum of two samples we need to understand the shape of the entire distribution, not just its first few moments. However, we have two forms of evidence that seem to support this conjecture: analyses based on approximations to the response time distribution, and simulations of the queueing model.

The primary approximation that we use is a recent result by Myers and Vernon [23] that only depends on the first two moments of the service time distribution. The approximation seems to perform fairly well in numerical evaluations with light-tailed service time distributions, such as the Erlang and hyperexponential distributions (see Figure 2 in [23]), although no bounds on the approximation error are available. However, the authors note that the approximation is likely to be inappropriate when the service times are heavy tailed.

As a supplement, therefore, in the heavy-tailed case, we use an approximation by Olvera-Cravioto et al. [24] that is applicable when the service times are regularly varying[1]. Heavy-tail approximations are fairly well established in queueing theory (see [26, 33]); the result due to Olvera-Cravioto et al. is, to the best of our knowledge, the most recent (and most accurate) refinement.

The following theorems summarize our results for these approximations. We omit the proofs due to space constraints.

Theorem 2. *Within the independence approximation and the approximation of the response time distribution due to Myers and Vernon [23], the threshold load is minimized when the service time distribution is deterministic.*

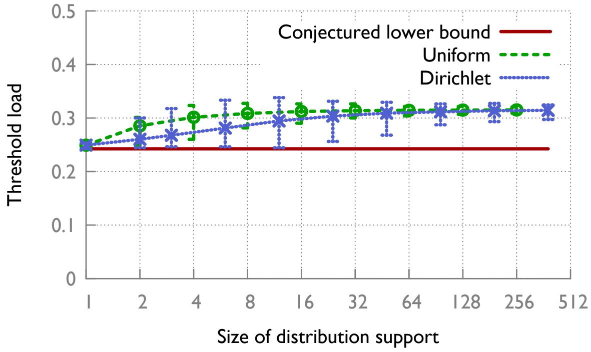


Figure 3: Randomly chosen service time distributions

The heavy-tail approximation by Olvera-Cravioto et al. [24] applies to arbitrary regularly varying service time distributions, but for our analysis we add an additional assumption requiring that the service time be sufficiently heavy. Formally, we require that the service time distribution have a higher coefficient of variation than the exponential distribution, which amounts to requiring that the tail index α be $< 1 + \sqrt{2}$. (The tail index is a measure of how heavy a distribution is: lower indices mean heavier tails.)

Theorem 3. *Within the independence approximation and the approximation due to Olvera-Cravioto et al. [24], if the service time distribution is regularly varying with tail index $\alpha < 1 + \sqrt{2}$, then the threshold load is $> 30\%$.*

Simulation results also seem to support the conjecture. We generated a range of service time distributions by, for various values of S, sampling from the space of all unit-mean discrete probability distributions with support $\{1, 2, ..., S\}$ in two different ways — uniformly at random, and using a symmetric Dirichlet distribution with concentration parameter 0.1 (the Dirichlet distribution has a higher variance and generates a larger spread of distributions than uniform sampling). Figure 3 reports results when we generate a 1000 different random distributions for each value of S and look at the minimum and maximum observed threshold load over this set of samples.

[1] The class of regularly varying distributions is an important subset of the class of heavy-tailed distributions that includes as its members the Pareto and the log-Gamma distributions.

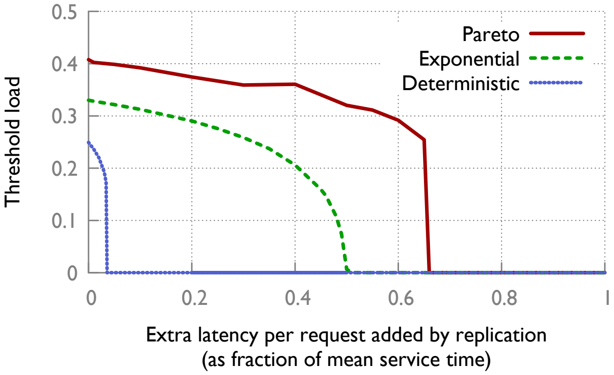


Figure 4: Effect of redundancy-induced client-side latency overhead, with different server service time distributions.

Effect of client-side overhead

As we noted earlier, our analysis so far assumes that the client-side overhead (e.g. added CPU utilization, kernel processing, network overhead) involved in processing the replicated requests is negligible. This may not be the case when, for instance, the operations in question involve large file transfers or very quick memory accesses. In both cases, the client-side latency overhead involved in processing an additional replicated copy of a request would be comparable in magnitude to the server latency for processing the request. This overhead can partially or completely counteract the latency improvement due to redundancy. Figure 4 quantifies this effect by considering what happens when replication adds a fixed latency penalty to every request. These results indicate that the more variable distributions are more forgiving of overhead, but client side overhead must be at least somewhat smaller than mean request latency in order for replication to improve *mean* latency. This is not surprising, of course: if replication overhead equals mean latency, replication cannot improve mean latency for any service time distribution — though it may still improve the tail.

2.2 Application: disk-backed database

Many data center applications involve the use of a large disk-based data store that is accessed via a smaller main-memory cache: examples include the Google AppEngine data store [16], Apache Cassandra [10], and Facebook's Haystack image store [7]. In this section we study a representative implementation of such a storage service: a set of Apache web servers hosting a large collection of files, split across the servers via consistent hashing, with the Linux kernel managing a disk cache on each server.

We deploy a set of Apache servers and, using a light-weight memory-soaking process, adjust the memory usage on each server node so that around half the main memory is available for the Linux disk cache (the other half being used by other applications and the kernel). We then populate the servers with a collection of files whose total size is chosen to achieve a preset target cache-to-disk ratio. The files are partitioned across servers via consistent hashing, and two copies are stored of every file: if the primary is stored on server n, the (replicated) secondary goes to server $n+1$. We measure the response time when a set of client nodes generate requests according to identical Poisson processes. Each request downloads a file chosen uniformly at random from the entire collection. We only test read performance on a static data set; we do not consider writes or updates.

Figure 5 shows results for one particular web-server configuration, with

- Mean file size = 4 KB
- File size distribution = deterministic, 4 KB per file
- Cache:disk ratio = 0.1
- Server/client hardware = 4 servers and 10 clients, all identical single-core Emulab nodes with 3 GHz CPU, 2 GB RAM, gigabit network interfaces, and 10k RPM disks.

Disk is the bottleneck in the majority of our experiments – CPU and network usage are always well below peak capacity.

The threshold load (the maximum load below which replication always helps) is 30% in this setup — within the 25-50% range predicted by the queueing analysis. Redundancy reduces mean latency by 33% at 10% load and by 25% at 20% load. Most of the improvement comes from the tail. At 20% load, for instance, replication cuts 99th percentile latency in half, from 150 ms to 75 ms, and reduces 99.9th percentile latency 2.2×.

The experiments in subsequent figures (Figures 6-11) vary one of the above configuration parameters at a time, keeping the others fixed. We note three observations.

First, as long as we ensure that file sizes continue to remain relatively small, changing the mean file size (Figure 6) or the shape of the file size distribution (Figure 7) does not siginificantly alter the level of improvement that we observe. This is because the primary bottleneck is the latency involved in locating the file on disk — when file sizes are small, the time needed to actually load the file from disk (which is what the specifics of the file size distribution affect) is negligible.

Second, as predicted in our queueing model (§2.1), increasing the variability in the system causes redundancy to perform better. We tried increasing variability in two different ways — increasing the proportion of access hitting disk by reducing the cache-to-disk ratio (Figure 8), and running on a public cloud (EC2) instead of dedicated hardware (Figure 9). The increase in improvement is relatively minor, although still noticeable, when we reduce the cache-to-disk ratio. The benefit is most visible in the tail: the 99.9th percentile latency improvement at 10% load goes up from 2.3× in the base configuration to 2.8× when we use the smaller cache-to-disk ratio, and from 2.2× to 2.5× at 20% load.

The improvement is rather more dramatic when going from Emulab to EC2. Redundancy cuts the mean response time at 10-20% load on EC2 in half, from 12 ms to 6 ms (compare to the $1.3-1.5\times$ reduction on Emulab). The tail improvement is even larger: on EC2, the 99.9th percentile latency at 10-20% load drops 8× when we use redundancy, from around 160 ms to 20 ms. It is noteworthy that the worst 0.1% of *outliers* with replication are quite close to the 12 ms *mean* without replication!

Third, as also predicted in §2.1, redundancy ceases to help when the client-side overhead due to replication is a significant fraction of the mean service time, as is the case when the file sizes are very large (Figure 10) or when the cache

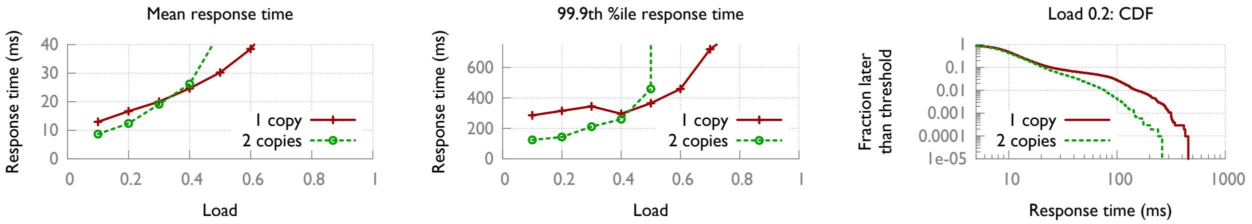


Figure 5: Base configuration

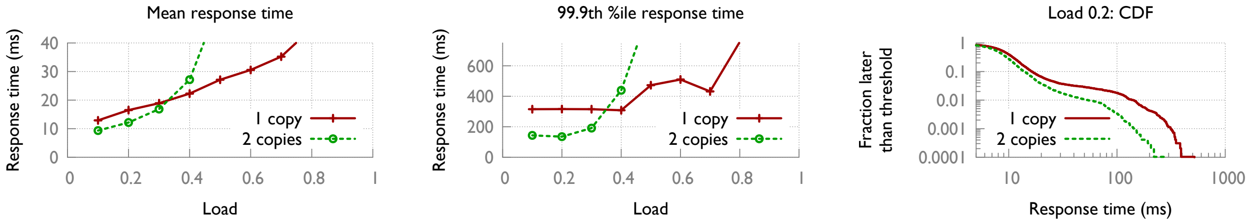


Figure 6: Mean file size 0.04 KB instead of 4 KB

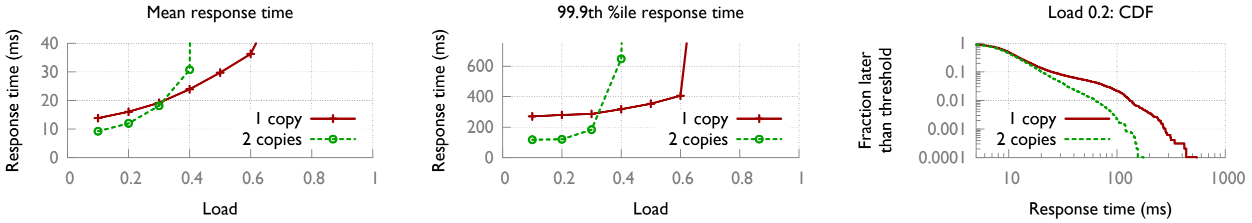


Figure 7: Pareto file size distribution instead of deterministic

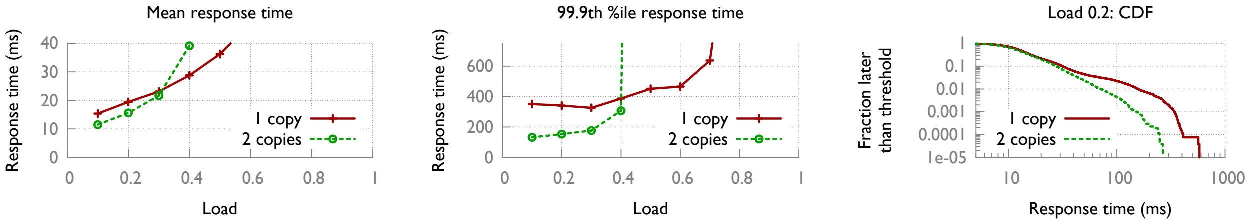


Figure 8: Cache:disk ratio 0.01 instead of 0.1. Higher variability because of the larger proportion of accesses hitting disk. Compared to Figure 5, 99.9th percentile improvement goes from 2.3× to 2.8× at 10% load, and from 2.2× to 2.5× at 20% load.

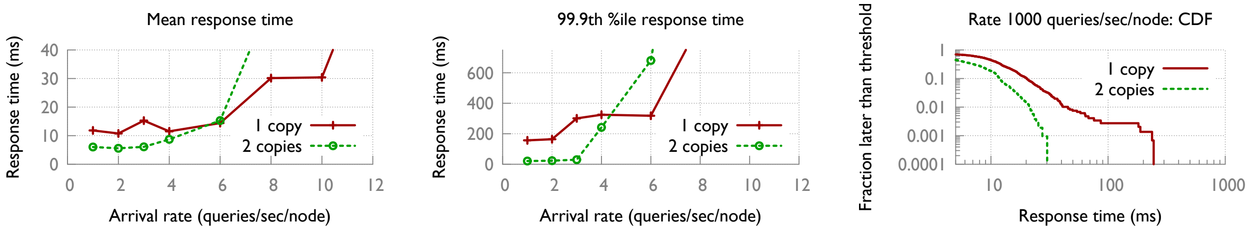


Figure 9: EC2 nodes instead of Emulab. x-axis shows unnormalised arrival rate because maximum throughput seems to fluctuate. Note the much larger tail improvement compared to Figure 5.

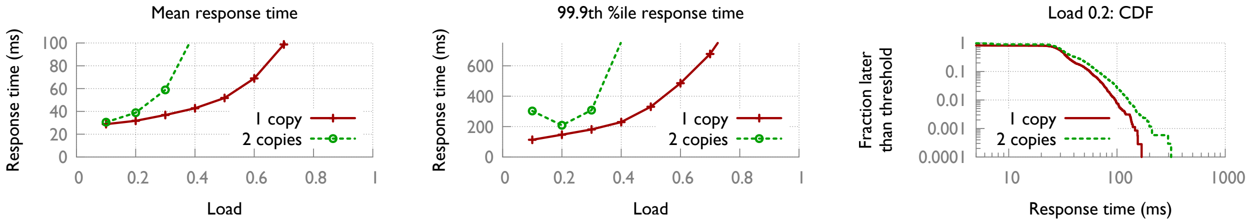


Figure 10: Mean file size 400 KB instead of 4 KB

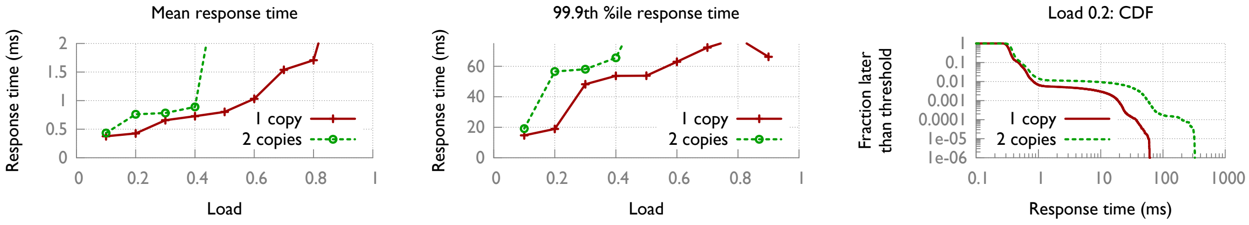


Figure 11: Cache:disk ratio 2 instead of 0.1. Cache is large enough to store contents of entire disk

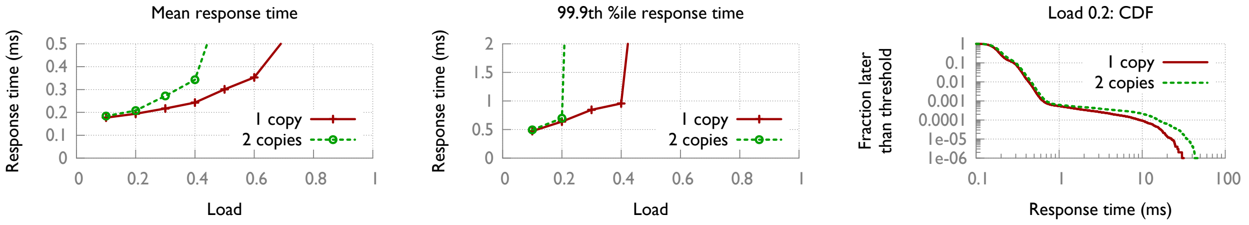


Figure 12: memcached

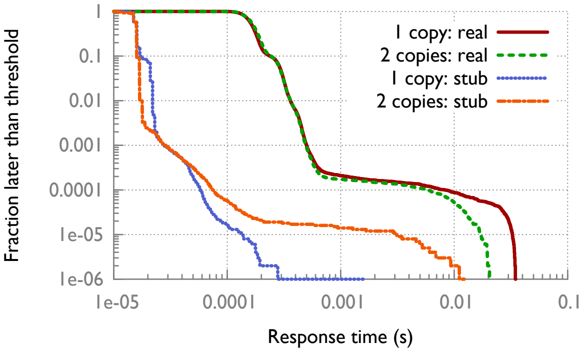


Figure 13: memcached: stub and normal version response times at 0.1% **load**

is large enough that all the files fit in memory (Figure 11). We study this second scenario more directly, using an in-memory distributed database, in the next section.

2.3 Application: memcached

We run a similar experiment to the one in the previous section, except that we replace the filesystem store + Linux kernel cache + Apache web server interface setup with the memcached in-memory database. Figure 12 shows the observed response times in an Emulab deployment. The results show that replication seems to worsen overall performance at all the load levels we tested (10-90%).

To understand why, we test two versions of our code at a low (0.1%) load level: the "normal" version, as well as a version with the calls to memcached replaced with stubs, no-ops that return immediately. The performance of this stub version is an estimate of how much client-side latency is involved in processing a query.

Figure 13 shows that the client-side latency is non-trivial. Replication increases the mean response time in the stub version by 0.016 ms, which is 9% of the 0.18 ms mean service time. This is an underestimate of the true client-side overhead since the stub version, which doesn't actually process queries, does not measure the network and kernel overhead involved in sending and receiving packets over the network.

The client-side latency overhead due to redundancy is thus at least 9% of the mean service time. Further, the service time distribution is not very variable: although there are outliers, more than 99.9% of the mass of the entire distribution is within a factor of 4 of the mean. Figure 4 in §2.1 shows that when the service time distribution is completely deterministic, a client-side overhead greater than 3% of the mean service time is large enough to completely negate the response time reduction due to redundancy.

In our system, redundancy does not seem to have that absolute a negative effect – in the "normal" version of the code, redundancy still has a slightly positive effect overall at 0.1% load (Figure 13). This suggests that the threshold load is positive though small (it has to be smaller than 10%: Figure 12 shows that replication always worsens performance beyond 10% load).

2.4 Application: replication in the network

Replication has always added a non-zero amount of overhead in the systems we have considered so far (even if that overhead was mitigated by the response time reduction it achieved). We now consider a setting in which this overhead can be essentially eliminated: a network in which the switches are capable of strict prioritization.

Specifically, we consider a data center network. Many data center network architectures [2, 18] provide multiple equal-length paths between each source-destination pair, and assign flows to paths based on a hash of the flow header [20]. However, simple static flow assignment interacts poorly with the highly skewed flow-size mix typical of data centers: the majority of the traffic volume in a data center comes from a small number of large elephant flows [2, 3], and hash-based flow assignment can lead to hotspots because of the possibility of assigning multiple elephant flows to the same link, which can result in significant congestion on that link. Recent work has proposed mitigating this problem by dynamically reassigning flows in response to hotspots, in either a centralized [1] or distributed [31] fashion.

We consider a simple alternative here: redundancy. Every switch replicates the first few packets of each flow along an alternate route, reducing the probability of collision with an elephant flow. Replicated packets are assigned a lower (strict) priority than the original packets, meaning they can never delay the original, unreplicated traffic in the network. Note that we could, in principle, replicate *every* packet — the performance when we do this can never be worse than without replication — but we do not since unnecessary replication can reduce the gains we achieve by increasing the amount of queueing *within* the replicated traffic. We replicate only the first few packets instead, with the aim of reducing the latency for short flows (the completion times of large flows depend on their aggregate throughput rather than individual per-packet latencies, so replication would be of little use).

We evaluate this scheme using an ns-3 simulation of a common 54-server three-layered fat-tree topology, with a full bisection-bandwidth fabric consisting of 45 6-port switches organized in 6 pods. We use a queue buffer size of 225 KB and vary the link capacity and delay. Flow arrivals are Poisson, and flow sizes are distributed according to a standard data center workload [8], with flow sizes varying from 1 KB to 3 MB and with more than 80% of the flows being less than 10 KB.

Figure 14 shows the completion times of flows smaller than 10 KB when we replicate the first 8 packets in every flow.

Figure 14(a) shows the reduction in the median flow completion time as a function of load for three different delay-bandwidth combinations (achieved by varying the latency and capacity of each link in the network). Note that in all three cases, the improvement is small at low loads, rises until load ≈ 40%, and then starts to fall. This is because at very low loads, the congestion on the default path is small enough that replication does not add a significant benefit, while at very high loads, every path in the network is likely to be congested, meaning that replication again yields limited gain. We therefore obtain the largest improvement at intermediate loads.

Note also that the performance improvement we achieve falls as the delay-bandwidth product increases. This is because our gains come from the reduction in queuing delay when the replicated packets follow an alternate, less congested, route. At higher delay-bandwidth products, queueing delay makes up a smaller proportion of the total flow

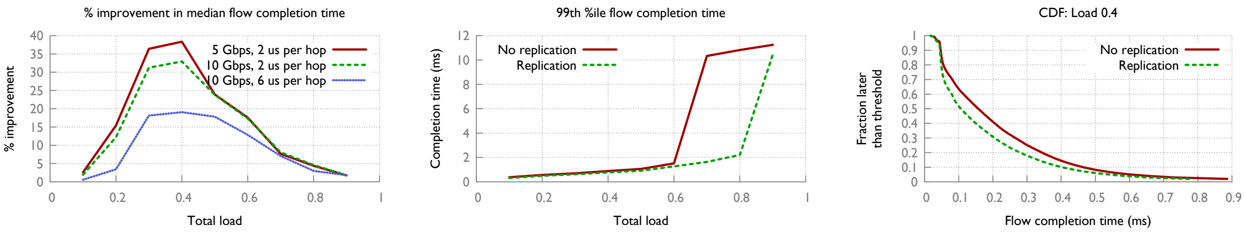


Figure 14: Median and tail completion times for flows smaller than 10 KB

completion time, meaning that the total latency savings achieved is correspondingly smaller. At 40% network load, we obtain a 38% improvement in median flow completion time (0.29 ms vs. 0.18 ms) when we use 5 Gbps links with 2 us per-hop delay. The improvement falls to 33% (0.15 ms vs. 0.10 ms) with 10 Gbps links with 2 us per-hop delay, and further to 19% (0.21 ms vs. 0.17 ms) with 10 Gbps links with 6 us per-hop delay.

Next, Figure 14(b) shows the 99th percentile flow completion times for one particular delay-bandwidth combination. In general, we see a 10-20% reduction in the flow completion times, but at 70-80% load, the improvement spikes to 80-90%. The reason turns out to be timeout avoidance: at these load levels, the 99th percentile unreplicated flow faces a timeout, and thus has a completion time greater than the TCP minRTO, 10 ms. With redundancy, the number of flows that face timeouts reduces significantly, causing the 99th percentile flow completion time to be much smaller than 10 ms.

At loads higher than 80%, however, the number of flows facing timeouts is high even with redundancy, resulting in a narrowing of the performance gap.

Finally, Figure 14(c) shows a CDF of the flow completion times at one particular load level. Note that the improvement in the mean and median is much larger than that in the tail. We believe this is because the high latencies in the tail occur at those instants of high congestion when most of the links along the flow's default path are congested. Therefore, the replicated packets, which likely traverse some of the same links, do not fare significantly better.

Replication has a negligible impact on the elephant flows: it improved the mean completion time for flows larger than 1 MB by a statistically-insignificant 0.12%.

3. INDIVIDUAL VIEW

The model and experiments of the previous section indicated that in a range of scenarios, latency is best optimized in a fixed set of system resources through replication. However, settings such as the wide-area Internet are better modeled as having *elastic* resources: individual participants can selfishly choose whether to replicate an operation, but this incurs an additional cost (such as bandwidth usage or battery consumption). In this section, we present two examples of wide-area Internet applications in which replication achieves a substantial improvement in latency. We argue that the latency reduction in both these applications outweighs the cost of the added overhead by comparing against a benchmark that we develop in a companion article [29]. The benchmark establishes a cost-effectiveness threshold by comparing the cost of the extra overhead induced at the servers and the clients against the economic value of the latency improvement that would be achieved. In our evaluation we find that the latency improvement achieved by redundancy is orders of magnitude larger than the required threshold in both the applications we consider here.

3.1 Application: Connection establishment

We start with a simple example, demonstrating why replication should be cost-effective even when the available choices are limited: we use a back-of-the-envelope calculation to consider what happens when multiple copies of TCP-handshake packets are sent on the same path. It is obvious that this should help if all packet losses on the path are independent. In this case, sending two back-to-back copies of a packet would reduce the probability of it being lost from p to p^2. In practice, of course, back-to-back packet transmissions are likely to observe a correlated loss pattern. But Chan et al. [11] measured a significant reduction in loss probability despite this correlation. Sending back-to-back packet pairs between PlanetLab hosts, they found that the average probability of individual packet loss was ≈ 0.0048, and the probability of *both* packets in a back-to-back pair being dropped was only ≈ 0.0007 – much larger than the $\sim 10^{-6}$ that would be expected if the losses were independent, but still 7× lower than the individual packet loss rate.[2]

As a concrete example, we quantify the improvement that this loss rate reduction would effect on the time required to complete a TCP handshake. The three packets in the handshake are ideal candidates for replication: they make up an insignificant fraction of the total traffic in the network, and there is a high penalty associated with their being lost (Linux and Windows use a 3 second initial timeout for SYN packets; OS X uses 1 second [12]). We use the loss probability statistics discussed above to estimate the expected latency savings on each handshake.

We consider an idealized network model. Whenever a packet is sent on the network, we assume it is delivered successfully after $(RTT/2)$ seconds with probability $1 - p$, and lost with probability p. Packet deliveries are assumed to be independent of each other. p is 0.0048 when sending one copy of each packet, and 0.0007 when sending two copies of each packet. We also assume TCP behavior as in the Linux kernel: an initial timeout of 3 seconds for SYN and SYN-ACK packets and of $3 \times RTT$ for ACK packets, and exponential backoff on packet loss [12].

With this model, it can be shown that duplicating all three packets in the handshake would reduce its expected comple-

[2] It might be possible to do even better by spacing the transmissions of the two packets in the pair a few milliseconds apart to reduce the correlation.

tion time by approximately $(3+3+3\times RTT)\times(4.8-0.7)$ ms, which is at least 25 ms. The benefit increases with RTT, and is even higher in the tail: duplication would improve the 99.9th percentile handshake completion time by at least 880 ms.

Is this improvement worth the cost of added traffic? Qualitatively, even 25 ms is significant relative to the size of the handshake packets. Quantitatively, a cost-benefit analysis is difficult since it depends on estimating and relating the direct and indirect costs of added traffic and the value to humans of lower latency. While an accurate comparison is likely quite difficult, the study referenced at the beginning of this section [29, 30] estimated these values using the pricing of cloud services, which encompasses a broad range of costs, including those for bandwidth, energy consumption, server utilization, and network operations staff, and concluded that in a broad class of cases, reducing latency is useful as long as it improves latency by 16 ms for every KB of extra traffic. In comparison, the latency savings we obtain in TCP connection establishment is more than an order of magnitude larger than this threshold in the mean, and more than two orders of magnitude larger in the tail. Specifically, if we assume each packet is 50 bytes long then a 25-880 ms improvement implies a savings of around 170-6000 ms/KB. We caution, however, that the analysis of [29, 30] was necessarily imprecise; a more rigorous study would be an interesting avenue of future work.

3.2 Application: DNS

An ideal candidate for replication is a service that involves small operations and which is replicated at multiple locations, thus providing diversity across network paths and servers, so that replicated operations are quite independent. We believe opportunities to replicate queries to such services may arise both in the wide area and the data center. Here, we explore the case of replicating DNS queries.

We began with a list of 10 DNS servers[3] and Alexa.com's list of the top 1 million website names. At each of 15 PlanetLab nodes across the continental US, we ran a two-stage experiment: (1) Rank all 10 DNS servers in terms of mean response time, by repeatedly querying a random name at a random server. Note that this ranking is specific to each PlanetLab server. (2) Repeatedly pick a random name and perform a random one of 20 possible trials — either querying one of the ten individual DNS servers, or querying anywhere from 1 to 10 of the best servers in parallel (e.g. if sending 3 copies of the query, we send them to the top 3 DNS servers in the ranked list). In each of the two stages, we performed one trial every 5 seconds. We ran each stage for about a week at each of the 15 nodes. Any query which took more than 2 seconds was treated as lost, and counted as 2 sec when calculating mean response time.

Figure 15 shows the distribution of query response times across all the PlanetLab nodes. The improvement is substantial, especially in the tail: Querying 10 DNS servers, the fraction of queries later than 500 ms is reduced by 6.5×, and the fraction later than 1.5 sec is reduced by 50×. Averaging over all PlanetLab nodes, Figure 16 shows the average percent reduction in response times compared to the best fixed DNS server identified in stage 1. We obtain a substantial reduction with just 2 DNS servers in all metrics, improving to 50-62% reduction with 10 servers. Finally, we compared performance to the best single server *in retrospect*, i.e., the server with minimum mean response time for the queries to individual servers in Stage 2 of the experiment, since the best server may change over time. Even compared with this stringent baseline, we found a result similar to Fig. 16, with a reduction of 44-57% in the metrics when querying 10 DNS servers.

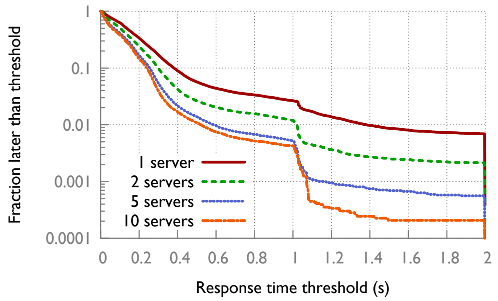


Figure 15: DNS response time distribution.

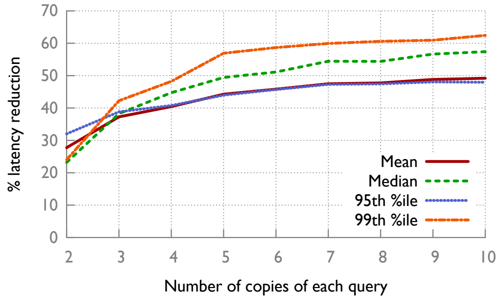


Figure 16: Reduction in DNS response time, averaged across 15 PlanetLab servers.

How many servers should one use? Figure 17 compares the marginal increase in latency savings from each extra server against the 16 ms/KB benchmark [29, 30] discussed earlier in this section. The results show that what we should do depends on the metric we care about. If we are only concerned with mean performance, it does not make economic sense to

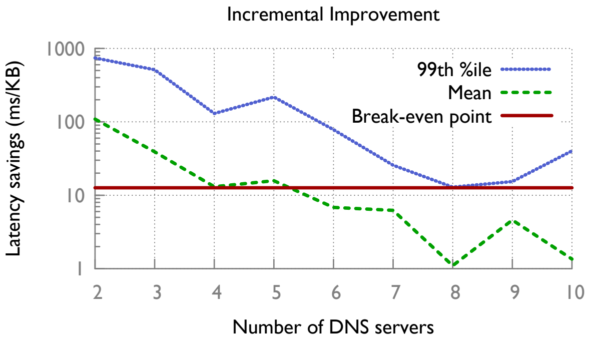


Figure 17: Incremental latency improvement from each extra server contacted

[3] The default local DNS server, plus public servers from Level3, Google, Comodo, OpenDNS, DNS Advantage, Norton DNS, ScrubIT, OpenNIC, and SmartViper.

contact any more than 5 DNS servers for each query, but if we care about the 99th percentile, then it is always useful to contact 10 or more DNS servers for every query. Note also that the *absolute* (as opposed to the marginal) latency savings is still worthwhile, even in the mean, if we contact 10 DNS servers for every query. The absolute mean latency savings from sending 10 copies of every query is 0.1 sec / 4500 extra bytes $\approx$ 23 ms/KB, which is more than twice the break-even latency savings. And if the client costs are based on DSL rather than cell service, the above schemes are all more than 100× more cost-effective.

Querying multiple servers also increases caching, a side-benefit which would be interesting to quantify.

Prefetching — that is, preemptively initiating DNS lookups for all links on the current web page — makes a similar tradeoff of increasing load to reduce latency, and its use is widespread in web browsers. Note, however, that redundancy is complementary to prefetching, since some names in a page will not have been present on the previous page (or there may not be a previous page).

4. RELATED WORK

Replication is used pervasively to improve reliability, and in many systems to reduce latency. Distributed job execution frameworks, for example, have used task replication to improve response time, both preemptively [4, 15] and to mitigate the impact of stragglers [32].

Within networking, replication has been explored to reduce latency in several specialized settings, including replicating DHT queries to multiple servers [22] and replicating transmissions (via erasure coding) to reduce delivery time and loss probability in delay-tolerant networks [21, 27]. Replication has also been suggested as a way of providing QoS prioritization and improving latency and loss performance in networks capable of redundancy elimination [19].

Dean and Barroso [13] discussed Google's use of redundancy in various systems, including a storage service similar to the one we evaluated in §2.2, but they studied specific systems with capabilities that are not necessarily available in general (such as the ability to cancel outstanding partially-completed requests), and did not consider the effect the total system utilization could have on the efficacy of redundancy. In contrast, we thoroughly evaluate the effect of redundancy at a range of loads both in various configurations of a deployed system (§2.2, §2.3), and in a large space of synthetic scenarios in an abstract system model (§2.1).

Andersen et al. [5]'s MONET system proxies web traffic through an overlay network formed out of multi-homed proxy servers. While the primary focus of [5] is on adapting quickly to changes in path performance, they replicate two specific subsets of their traffic: connection establishment requests to multiple servers are sent in parallel (while the first one to respond is used), and DNS queries are replicated to the local DNS server on each of the multi-homed proxy server's interfaces. We show that replication can be useful in both these contexts even in the absence of path diversity: a significant performance benefit can be obtained by sending multiple copies of TCP SYNs to the *same* server on the *same* path, and by replicating DNS queries to multiple public servers over the *same* access link.

In a recent workshop paper [30] we advocated using redundancy to reduce latency, but it was preliminary work that did not characterize when redundancy is helpful, and did not study the systems view of optimizing a fixed set of resources.

Most importantly, unlike all of the above work, our goal is to demonstrate the power of redundancy as a general technique. We do this by providing a characterization of when it is (and isn't) useful, and by quantifying the performance improvement it offers in several use cases where it is applicable.

5. CONCLUSION

We studied an abstract characterization of the tradeoff between the latency reduction achieved by redundancy and the cost of the overhead it induces to demonstrate that redundancy should have a net positive impact in a large class of systems. We then confirmed empirically that redundancy offers a significant benefit in a number of practical applications, both in the wide area and in the data center. We believe our results demonstrate that redundancy is a powerful technique that should be used much more commonly in networked systems than it currently is. Our results also will guide the *judicious* application of redundancy within only those cases where it is a win in terms of performance or cost-effectiveness.

Acknowledgements

We would like to thank our shepherd Sem Borst and the anonymous reviewers for their valuable suggestions. We gratefully acknowledge the support of NSF grants 1050146, 1149895, 1117161 and 1040838.

6. REFERENCES

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] M. Alizadeh, S. Yang, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. Deconstructing datacenter packet transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 133–138, New York, NY, USA, 2012. ACM.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Why let resources idle? Aggressive cloning of jobs with Dolly. In *USENIX HotCloud*, 2012.

[5] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao. Improving web availability for clients with MONET. In *USENIX NSDI*, pages 115–128, Berkeley, CA, USA, 2005. USENIX Association.

[6] S. Asmussen. *Applied Probability and Queues*. Wiley, 1987.

[7] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, pages 267–280, New York, NY, USA, 2010. ACM.

[9] J. Brutlag. Speed matters for Google web search, June 2009. http://services.google.com/fh/files/blogs/google_delayexp.pdf.

[10] Apache Cassandra. http://cassandra.apache.org.

[11] E. W. Chan, X. Luo, W. Li, W. W. Fok, and R. K. Chang. Measurement of loss pairs in network paths. In *IMC*, pages 88–101, New York, NY, USA, 2010. ACM.

[12] J. Chu. Tuning TCP parameters for the 21st century. http://www.ietf.org/proceedings/75/slides/tcpm-1.pdf, July 2009.

[13] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[14] P. Dixon. Shopzilla site redesign – we get what we measure, June 2009. http://www.slideshare.net/shopzilla/shopzillas-you-get-what-you-measure-velocity-2009.

[15] C. C. Foster and E. M. Riseman. Percolation of code to enhance parallel dispatching and execution. *IEEE Trans. Comput.*, 21(12):1411–1415, Dec. 1972.

[16] Google AppEngine datastore: memcached cache. https://developers.google.com/appengine/docs/python/memcache/usingmemcache#Pattern.

[17] W. Gray and D. Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied*, 6(4):322, 2000.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. In *ACM SIGCOMM*, pages 51–62, New York, NY, USA, 2009. ACM.

[19] D. Han, A. Anand, A. Akella, and S. Seshan. RPT: re-architecting loss protection for content-aware networks. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[20] C. Hopps. Computing TCP's retransmission timer (RFC 6298), 2000.

[21] S. Jain, M. Demmer, R. Patra, and K. Fall. Using redundancy to cope with failures in a delay tolerant network. In *ACM SIGCOMM*, 2005.

[22] J. Li, J. Stribling, R. Morris, and M. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *NSDI*, 2005.

[23] D. S. Myers and M. K. Vernon. Estimating queue length distributions for queues with random arrivals. *SIGMETRICS Perform. Eval. Rev.*, 40(3):77–79, Jan. 2012.

[24] M. Olvera-Cravioto, J. Blanchet, and P. Glynn. On the transition from heavy-traffic to heavy-tails for the m/g/1 queue: The regularly varying case. *Annals of Applied Probability*, 21:645–668, 2011.

[25] S. Ramachandran. Web metrics: Size and number of resources, May 2010. https://developers.google.com/speed/articles/web-metrics.

[26] K. Sigman. Appendix: A primer on heavy-tailed distributions. *Queueing Systems*, 33(1-3):261–275, 1999.

[27] E. Soljanin. Reducing delay with coding in (mobile) multi-agent information transfer. In *Communication, Control, and Computing (Allerton), 2010 48th Annual Allerton Conference on*, pages 1428–1433. IEEE, 2010.

[28] S. Souders. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html.

[29] A. Vulimiri, P. B. Godfrey, and S. Shenker. A cost-benefit analysis of low latency via added utilization, June 2013. http://web.engr.illinois.edu/~vulimir1/benchmark.pdf.

[30] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is less: Reducing latency via redundancy. In *Eleventh ACM Workshop on Hot Topics in Networks (HotNets-XI)*, October 2012.

[31] X. Wu and X. Yang. Dard: Distributed adaptive routing for datacenter networks. In *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 32–41, Washington, DC, USA, 2012. IEEE Computer Society.

[32] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *USENIX OSDI*, pages 29–42, Berkeley, CA, USA, 2008.

[33] A. P. Zwart. *Queueing Systems With Heavy Tails.* PhD thesis, Technische Universiteit Eindhoven, September 2001.