2009 28th IEEE International Symposium on Reliable Distributed Systems

When and How to Change Quorums on Wide Area Networks

Michael G. Merideth, Florian Oprea
Carnegie Mellon University
Pittsburgh, PA, USA

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA

***Abstract*—In wide-area settings, unpredictable events, such as flash crowds caused by nearly instantaneous popularity of services, can cause servers that are expected to respond quickly to instead suddenly respond slowly. This presents a problem for achieving consistently good performance in quorum-based distributed systems, in which clients must choose which quorums (sets of servers) to access. Typically, clients are motivated to choose quorums containing the servers that respond fastest. Often, these may be the closest servers, but when the closest servers are particularly slow to respond, e.g., because of a changed workload, servers that are farther may actually respond faster. In this paper, we show how clients can locally change their quorum selections efficiently such that the overall system performance rapidly converges to that of the best global strategy for the current conditions. Moreover, we discuss how to benefit even when changes in quorums must be accompanied by expensive state-transfer operations.**

***Keywords*-Quorum system; wide-area network; access strategy; dynamic workload**

I. Introduction

A *quorum system* is a collection of sets (*quorums*) of servers, such that any two quorums have non-empty intersection. Quorum systems are a standard tool for improving the efficiency and fault tolerance of a distributed system. In a typical quorum-based application, a read or update operation is performed by contacting all of the servers of some quorum (possibly in more than one round of interaction). The intersection property enables a read or update operation to observe the effects of prior update operations. Moreover, the fact that not all servers need to be contacted for each operation can increase efficiency if the processing load is dispersed across servers through the use of different quorums. These properties of quorum systems make them the tool of choice for a number of practical protocol implementations (e.g., [1], [2], [3]).

In many respects, quorum systems are well-suited to wide-area networks in which network delays to reach servers can be varied and significant. In contrast to replication schemes that require every server to be contacted, quorum systems can work without the participation of the most distant servers. However, achieving good performance on a wide-area network requires consideration of the workload imposed on the servers. As a result, as illustrated qualitatively by Figure 1, static quorum selection solutions are not well-suited to changing workloads. Intuitively, a client should use a closest strategy, i.e., select a nearby quorum for its operation, if those servers are lightly loaded (so as to minimize network delays), as has been previously validated [4]. However, if nearby quorums are too busy, which can happen, for example, as a result of changes in the distribution of traffic, clients may be better served by using a balanced strategy making use of less loaded quorums that are farther away [4].

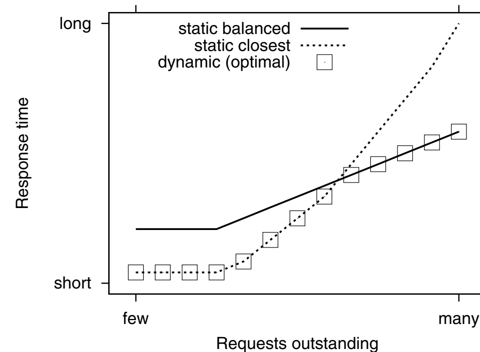


Figure 1. Dynamic quorum selection compared with static selection.

Wide-area networks are often subject to unpredictable or changing workloads due, e.g., to flash-crowd events or diurnal patterns of activity [5], [6], [7]. Yet, to our knowledge, the problem of designing load-dispersing algorithms for quorum systems that can react to these changes has not been considered before (see Section II for a discussion of related work). To address this, we propose two algorithms in Section IV by which clients can locally change their quorum selections efficiently such that the overall system performance rapidly converges to that of the best global strategy for the current conditions. Each client uses only information it gleans from previous operation invocations to the service without communicating with other clients. Moreover, these algorithms impose negligible overhead on servers.

Our algorithms in Section IV are general in the sense that they make no assumptions about the specifics of the quorum system protocol being employed. However, a particularly important use of quorums is in the maintenance of replicated state (e.g., [8], [9], [1], [2], [10]). In such systems, a switch to a new quorum may result in the need to transfer state to the new quorum from the old. This can be an expensive

1060-9857/09 $26.00 © 2009 IEEE
DOI 10.1109/SRDS.2009.35

IEEE computer society

operation on a wide area network. Thus, in Section V, we propose and evaluate an alternative way to balance the load across the servers when state transfer is a concern.

II. Related work

In the past, researchers have made considerable efforts to improve the performance of quorum-based protocols in wide-area networks (WANs), both via foundational studies (e.g., [11], [12], [13], [14], [15], [16]) and with practical protocols (e.g., [17], [18], [4]). Yet, to our knowledge, only our own prior work [4] considers the problem of optimizing the performance of quorum systems on WANs through the choice of quorums. In [4], our approaches minimize average response times under fixed workloads (i.e., assuming no flash crowds or diurnal effects). The approaches are based on linear programs that make use of statistics from the entire network that are assumed not to change. Thus, the algorithms in [4] can neither react to changing workloads nor be run by a client in isolation with a limited view of the system. In contrast, in the current paper, we present low-overhead algorithms that are suitable for allowing clients to change their quorum selections efficiently based on limited knowledge. These algorithms can react to changing workloads while producing solutions that lead to very good system performance. Moreover, unlike [4], this paper considers the costs of state transfer.

Although dynamic quorum selection on WANs has not been studied before, choosing a *single* server to optimize client response time on WANs is a well-studied problem. The solutions proposed range from selecting servers based only on distance from clients [19], [20], [21] or only on load at servers [22], to using both network delay and load at servers when choosing the best server [23]. From this perspective, our work is more closely related to the latter since both network delay and server load contribute to the response time estimates that we use in deciding how to access quorums.

Balancing load for quorum systems in general (i.e., without accounting for network delays) is also a fairly well-studied problem. Naor and Wool [24] define a formal notion of load and developed load-optimal quorum constructions and *access strategies* (probability distributions on quorums from which clients sample quorums to perform accesses; see Section III) to achieve optimal load for these constructions. Malkhi et al. [25], [26] extend this treatment to quorum systems with stronger intersection requirements that enable quorums to be used in systems that may suffer Byzantine server failures; the Q/U access protocol [1], which we use in Section V, is designed for this fault model. Holzman et al. [27] examine quorum systems from the perspective of the *load balancing ratio*: the ratio between the most and the least loaded server of a quorum system. They describe methods for determining the access strategies for which a quorum system achieves optimal balancing.

Some prior work [13], [12], [18], [28] studies other ways to optimize performance of quorum-based systems on WANs. Gao et al. [17] combine volume leases and quorum systems into a protocol for replicated edge-services that performs well for workloads exhibiting certain locality properties. Amir et al. [18] construct an efficient Byzantine-fault-tolerant replicated service for WANs using the state-machine-replication protocol Paxos [29]. Mao et al. [30] derive a new version of Paxos that partitions sequence numbers amongst multiple leaders to achieve higher throughput under load and lower latencies for clients. However, none of these shows when and how to choose quorums to adapt to dynamic workloads on WANs.

III. Preliminaries

A quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ over a universe U of servers is a collection of subsets of U such that any two have non-empty intersection. In applications that use quorum systems, a client needs to contact a quorum of servers for each request. To choose the quorum to access, the client c samples a quorum according to a distribution p^c on quorums, i.e., $\sum_{Q_i \in \mathcal{Q}} p^c(Q_i) = 1$ for each client c. p^c is also known as the *access strategy* for client c.

Our algorithms are designed for applications that employ quorum systems over a wide-area network. In this environment, communication between different nodes incurs an inherent cost—the network latency between the nodes. The algorithms we propose rely on each client estimating the network latency to reach each server at coarse time intervals. This can be easily achieved by means of round-trip-time measurements or using other existing mechanisms [31], [32]. We denote the network delay from a client c to a server s by $d(c, s)$, and we denote by $d(c, Q) = \max_{s \in Q} d(c, s)$ the delay for c to access quorum $Q \in \mathcal{Q}$. Note that since $d(c, Q) = \max_{s \in Q} d(c, s)$, we presume that c accesses the servers in Q concurrently, not sequentially.

Two access strategies are known to perform well for certain workloads on wide area networks [4]. One, denoted $\mathsf{ClosestDly}$, is to access the closest quorum in terms of static network latency. In other words, for each client c, $\mathsf{ClosestDly}^c$ assigns equal probability to each quorum that is closest to c, and $\mathsf{ClosestDly}^c(Q) = 0$ for all other quorums Q. $\mathsf{ClosestDly}$ works well when the workload is sufficiently light that the processing load imposed on servers contributes a negligible amount to the response times observed by clients. The second strategy, denoted $\mathsf{Balanced}$, causes each client to access quorums uniformly at random, thereby balancing load on servers, assuming that each server is in the same number of quorums (which holds for all quorum systems considered here). That is, for each client c, $\mathsf{Balanced}^c$ assigns the same probability to each $Q \in \mathcal{Q}$. This access strategy makes sense when the workload is sufficiently heavy that response times are overwhelmingly

dominated by processing delays, which are best spread uniformly across servers. (For an exception, see Section V.)

To our algorithms, a failed server is one that exhibits very high response time; whether this is due to failure or simply very high load is irrelevant to our algorithms. As such, we do not treat failures separately in our evaluations.

IV. Dynamic quorum selection algorithms

A *dynamic quorum selection algorithm* updates quorum selections based on changes in response times. We allow that clients may have limited views of these changes and will not always have the most up-to-date information about all servers. Therefore, our algorithms require clients to maintain *estimates* of response times based on their recent history of requests. We begin our discussion with the presentation of a naive algorithm, ClosestRT, for dynamically updating the strategy for quorum selection. Then, in the following subsections, we present more sophisticated algorithms—ShiftWt (and its variants) and DelayBins—that perform better than ClosestRT. However, since our evaluation is largely empirical, we first present its setup.

A. Evaluation setup

Throughout this section, we evaluate our algorithms using two generic quorum-based operation-invocation protocols: one that uses a $k \times k$ Grid quorum system [33], [34], and one that uses a $(4b+1, 5b+1)$ Majority [35], [36] quorum system. In a $k \times k$ Grid, the system contains k^2 servers, which are logically organized into a grid with k rows and k columns. A quorum is comprised of the servers in the union of any row and any column, and so the number of quorums is also $m = k^2$. The $(4b+1, 5b+1)$ Majority consists of all the subsets of size $4b+1$ of a set of $5b+1$ of servers. This type of quorum system has been in used in practical Byzantine-fault-tolerant state-machine-replication protocols such as Q/U [1]. The b parameter represents the number of Byzantine server failures tolerated. While the Grid quorum system highlights the differences between the strategies because of its small quorum sizes relative to the total number of servers, we also include results for the Majority quorum system where instructive. Additional similar results can be found in [37], but are omitted here for brevity.

We evaluate our algorithms with respect to several measures. *Convergence time* measures how fast our algorithms converge to the optimal access strategy for a certain workload after a change in the workload. To measure convergence time, we initialize clients with access strategies of Balanced in a regime of low load and measure the time it takes for clients to transform their access strategies into ClosestDly. We say that the algorithm has converged to ClosestDly when the L2 distance between $\mathsf{avg}\{\mathsf{ClosestDly}^c\}_c$ and the average $\mathsf{avg}\{p^c\}_c$ of the per-client access strategies determined by the algorithm is less than 0.01. We perform the converse experiment as well: we initialize our algorithm to ClosestDly in a regime of high load and measure the time it takes for clients to change their access strategies to Balanced. *Response time* is computed by averaging the response times observed by all clients, where the response time of a single client is the average of all requests made in a run. *Throughput* is obtained by summing the request throughput of all clients. Finally, we also measure the *overhead* associated with each algorithm, where overhead is the average time per request spent in the quorum selection function.

Our evaluation in this section employs a generic wide-area topology that is derived from PlanetLab [38] measurements and emulated using Modelnet [39]. We deploy Modelnet on a rack of 76 Intel Pentium 4 2.80 GHz computers, each with 1 GB of memory and an Intel PRO/1000 NIC. We use one machine as the WAN emulator and the remaining 75 as normal hosts (servers or clients). The topology consists of 50 nodes and the latencies between them as measured between distinct sites on PlanetLab (see http://ping.ececs.uc.edu/ping/). This matches our focus on WANs since 93% of all pairs of nodes in the dataset have a latency of at least 20 milliseconds (ms) between them. The average round-trip time between pairs of nodes is about 107 ms with a standard deviation of 73 ms, while the maximum distance between any two nodes is 386 ms. Other than latency, we set no constraints on the links of our emulated topology (no bandwidth constraints, for instance). Repeated experiments on a different topology of 50 other PlanetLab sites provide qualitatively similar results [37], and so we omit them here for brevity.

In any given experiment, we place servers onto nodes of this topology in a one-to-one fashion (i.e., any node hosts at most one server) using algorithms presented by Gupta et al. [13]. These placement algorithms minimize the average, over all nodes, of each node c's expected network delay when accessing quorums using $\mathsf{Balanced}^c$. To generate load we choose 10 nodes uniformly at random, and start clientsPerNode clients on each node, with clientsPerNode ranging from 1 to a maximum number sufficient to saturate the system (i.e., to achieve maximum throughput). Each client issues requests sequentially, i.e., issuing a new request immediately when its previous request completes. All requests complete in a single round trip. We set the application processing time per request to 1 ms.

B. ClosestRT*: A naive algorithm*

Given response time estimates for all servers, the most naive algorithm for selecting a quorum for a request is to choose the quorum with the lowest observed response time. We call this algorithm ClosestRT. In our implementation, each client keeps the response times it has observed for each server s in the last $history_duration$ seconds, along with timestamps indicating when these observations were made. Assuming these (response time, timestamp) pairs are $(rt^c_i(s), ts^c_i(s))$, for $i = 1 \ldots t$, the client computes the

current estimate $rt^c(s)$ at the current time ts, as $rt^c(s) = \sum_{i=1}^{t} w(ts_i^c(s), ts) rt_i^c(s)$, i.e., as a non-uniform average of the response times seen in the past from this server. The values $w(ts_i^c(s), ts)$ are non-negative reals and satisfy $\sum_{i=1}^{t} w(ts_i^c(s), ts) = 1$. They are chosen such that response times of more recent requests have more influence on $rt^c(s)$ (i.e., we favor more recent requests when computing the estimate). If c has not accessed s in the last $history_duration$ seconds, it sets $rt^c(s)$ equal to its static round-trip network latency to s, which is an (optimistic) estimate of its response time.

The accuracy of this estimation method depends on the $history_duration$ parameter. For example, setting it too close to 0 shrinks the number of servers for which client c has some indication of load to only the quorum used in the last access. To avoid this problem, we use a sufficiently large value for $history_duration$ ($history_duration$ = 1 second performs well in our experiments).

We evaluate ClosestRT against ClosestDly and Balanced on a 5×5 Grid quorum system ranging the number of clients on each host from 1 to 15. Figure 3(a) shows that ClosestRT performs only 10 ms to 15 ms worse than ClosestDly in case of low load, but up to 40 ms worse than Balanced in case of high load. The most likely cause for this decay in performance as load increases is the so-called "herd effect": clients with similar response-time estimates synchronize and switch to the same quorum, i.e., the quorum having the smallest response time. In doing so they lose the load balancing benefits of Balanced. One can fix the herd effect problem by shifting only a fraction of the load (say, half) to a different quorum when changing a client's access strategy. We discuss this option next.

C. ShiftWt: *A weight shifting algorithm*

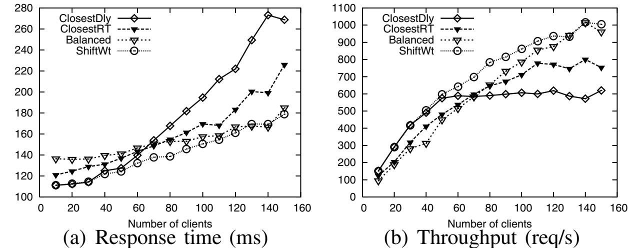
Our next algorithm, ShiftWt, uses the following simple strategy for updating a client's access strategy: each client c starts with an arbitrary access strategy p^c and periodically shifts half of the weight from the quorum Q_j with the highest response time (i.e., half of $p^c(Q_j)$) to the quorum with the lowest response time. Note that a client c cannot shift any weight from a quorum Q_j with $p^c(Q_j) = 0$. Also note that there can be more than one quorum with the highest response time; in fact, all quorums containing the server or servers with the highest response time also have the highest response time. To balance load faster, we select the quorum from which we shift weight as follows: sort the servers in decreasing order by response time to obtain $rt^c(U) = \langle rt^c(s) \rangle_{s \in U}$ and let $rt^c(Q)$ be the projection of $rt^c(U)$ onto Q (see Figure 2 for exact definition). Sort all nonzero weight quorums in decreasing lexicographical order by $rt^c(Q)$ and pick the first as the quorum from which to shift weight.

Figure 2 presents a pseudocode description of ShiftWt. In it, we set the initial access strategy to Balanced in the initialization phase, since Balanced is optimal for high load. One can start from other access strategies and obtain good performance, but as we will see, the convergence time to an optimal access strategy for a certain load depends on the access strategy used initially.

INITIALIZATION:
for $i = 1$ to m
  $p^c(Q_i) = 1/m$;
BEFORE EACH REQUEST:
for each $s \in U$, estimate $rt^c(s)$;
sort servers in decreasing order by response time
  into $rt^c(U) = \langle rt^c(s) \rangle_{s \in U}$
for each quorum Q define $rt^c(Q)$ as:
  $rt^c(Q)[s] = rt^c(U)[s]$ for $s \in Q$
  $rt^c(Q)[s] = 0$ for $s \in U \setminus Q$
sort quorums lexicographically by $rt^c(Q)$ in decreasing order and
  let Q_j be the highest non-zero weight quorum
  let Q_i be the lowest weight quorum
update the client's access strategy as follows:
  $p^c_{\mathsf{new}}(Q_j) = p^c(Q_j)/2$;
  $p^c_{\mathsf{new}}(Q_i) = p^c(Q_i) + p^c(Q_j)/2$;
  $p^c_{\mathsf{new}}(Q_k) = p^c(Q_k)$, for $k \neq i, j$;
  $p^c = p^c_{\mathsf{new}}$;

Figure 2. ShiftWt algorithm at client c

In Figure 3, we compare ShiftWt to ClosestDly, ClosestRT and Balanced for the Grid quorum system. ShiftWt always performs better than ClosestRT, and as well as ClosestDly and Balanced in load regimes when these are best (low-load and high-load respectively). In fact, when ClosestDly and Balanced perform comparably well (between 60 and 90 clients in total) ShiftWt is better than both. This holds true for both response time and throughput. Note though that throughput here is limited by the 1 ms processing time per request at each server.



(a) Response time (ms) (b) Throughput (req/s)

Figure 3. Comparison between strategies

Unfortunately, the convergence time of ShiftWt is less attractive in our experiments. ShiftWt takes 48 seconds to converge to ClosestDly in a setting with 10 clients (clientsPerNode = 1), and in a setting with 150 clients (clientsPerNode = 15) it never converges to Balanced, but rather converges (in 5 seconds) to an access strategy at distance 0.15 from Balanced. While this second convergence time is much better, the first convergence time is clearly too high, motivating us to seek ways to improve it.

A way to make ShiftWt converge faster is to shift more load every time the access strategy for a client is changed.

Instead of shifting more weight from a single quorum, we choose to shift load from more quorums at once. Clearly the quorums with negative impact on performance are the quorums with the highest response time. Thus we choose to shift load from all the quorums Q_i with positive weight (i.e., $p^c(Q_i) > 0$) containing the server with the highest response time. If no such quorums exist, we look at the quorums containing the server with the second-highest response time.

The question is where to place the load collected from the quorums with high response time. Putting it on a single quorum might unbalance the system and lead again to suboptimal performance. Ideally, we would add weight to the quorums with the smallest response times so that after absorbing this extra load, their response times would be roughly the same. To achieve this, we use the following simple algorithm. Let $Q_1, \ldots, Q_m$ denote the quorums in increasing order of their response times (i.e., in increasing order of $\max_{s \in Q_i}\{rt^c(s)\}$), and suppose we have to place total weight w on the l lowest-response-time quorums (i.e., on $Q_1, \ldots, Q_l$). We split the weight w into l weights w_i, for $i = 1, \ldots, l$, where w_i is proportional to $\max_{s \in Q_{l+1}}\{rt^c(s)\} - \max_{s \in Q_i}\{rt^c(s)\}$. Obviously this approach puts more weight on quorums with smaller response times, which is what we wanted. We choose l, the number of quorums to which we shift weight, equal to the number of quorums from which we took weight in the first place. We call the resulting algorithm ShiftWtOpt.

We compare the convergence time of ShiftWtOpt to that of ShiftWt. Figure 4(a) shows convergence to ClosestDly in a low-load regime while Figure 4(b) shows convergence to Balanced in a high-load regime. (Access strategies are fixed in the initial 10 seconds of each run, allowing for all clients to join the experiment.) Results show a convergence time to ClosestDly of about 8 seconds for ShiftWtOpt, a six-fold improvement compared to ShiftWt, and a convergence time to Balanced of about 2 seconds for ShiftWtOpt, a two-fold improvement compared to ShiftWt. Also, ShiftWtOpt gets closer to Balanced than does ShiftWt.

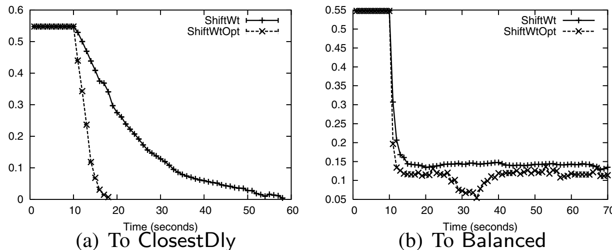


Figure 4. Convergence (L2 Norm)

The improvements of ShiftWtOpt over ShiftWt come at the cost of increased client overhead, however. Table I shows the client-side overhead in microseconds (μs) of ShiftWtOpt for different universe sizes. As Table I shows, the overhead increases with the number of quorums. While this value is perhaps inflated due to the fact that our implementation is not optimized, we unfortunately expect that no implementation of ShiftWtOpt can eliminate this increasing overhead as the system scales, since it has to at least examine all of the quorums containing the server with the highest response time.

k	$k \times k$ Grid	
	ShiftWtOpt	DelayBins
2	8.31 μs	1.64 μs
3	14.52 μs	1.69 μs
4	23.92 μs	1.82 μs
5	38.03 μs	1.92 μs

Table I
CLIENT OVERHEAD PER REQUEST

The results in Table I suggest that, for quorum systems with a small number of quorums, the client overhead is very good, especially considering the fact that any request that travels over the WAN takes at least several milliseconds, i.e., orders of magnitude longer than the overhead. However, in experiments with Majority quorum systems we have found dramatically larger overheads [37]. Further, we have experimented with several variations of ShiftWtOpt [37], but none perform much better than the one described here. In general ShiftWt variants face the following trade-off: better efficiency through simplicity (e.g., shifting weight between only two quorums), but increased convergence time; versus better convergence time through more aggressive quorum analysis (either when choosing the quorums or when choosing the number of pairs of quorums between which load is shifted), but additional overhead. These results provide sufficient motivation to seek a better quorum-selection algorithm as described next.

D. DelayBins*: Faster convergence.*

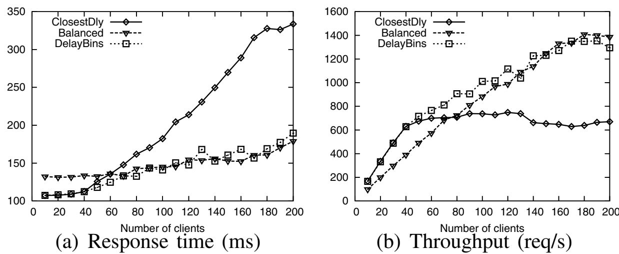
In our final algorithm in this section, DelayBins, each client c partitions the quorums into groups based on the network delay to reach them. Denote the groups by $G^c_1, \ldots, G^c_t$ and by $d^c_1 \leq \ldots \leq d^c_t$ the corresponding delays. Thus, $G^c_j = \{Q_i \mid d(c, Q_i) = d^c_j\}$. Let $H^c_j = \{Q_i \mid d(c, Q_i) \leq d^c_j\}$ be the collection of quorums at distance at most d^c_j from the client. Obviously, $H^c_j = G^c_1 \cup \ldots \cup G^c_j$. At any point in time the client chooses a quorum by sampling uniformly at random from some group $H^c_{curr_quorum_group}$ where $1 \leq curr_quorum_group \leq t$.

The client keeps an average avg_rt of the response times for its last $history_count$ requests. To choose the quorum group used for sampling, the client compares avg_rt with the maximum response time it would expect to see from any quorum in $H^c_{curr_quorum_group}$ if load were low. For instance, if a client currently uses group H^c_j, the response time it sees in case of low load should never be higher than d^c_j + $server_compute_time$ (here $server_compute_time$

denotes the application processing cost per request). When this happens, it indicates an increase in load at servers. The client however will not switch to the next higher group (H^c_{j+1}) unless $avg_rt \geq d^c_{j+1} + server_compute_time$. Setting the size of the $history_count$ parameter to different values allows one to trade-off accuracy of the average with how quickly the algorithm adapts to load changes. Since the algorithm partitions quorums into groups based on the distance to the client, we call it DelayBins. The pseudocode for DelayBins is presented in Figure 5.

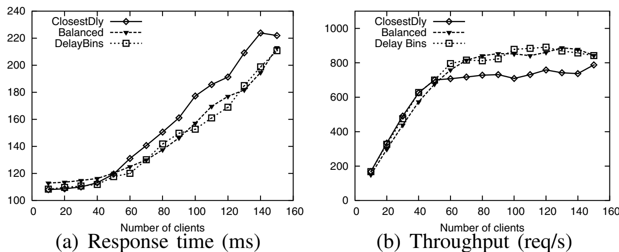
INITIALIZATION:
$curr_quorum_group = 1$; $last_req = 0$;
create quorum groups $H^c_1, \ldots, H^c_k$, $k \geq 1$;
corresponding delays are $d^c_1, \ldots, d^c_k$;
$d^c_0 = 0$; $d^c_{k+1} = \infty$;
BEFORE EACH REQUEST:
find j such that $avg_rt \in [d^c_j, d^c_{j+1}) + server_compute_time$;
if $j > curr_quorum_group$;
$curr_quorum_group++$;
if $j < curr_quorum_group$
$curr_quorum_group = j$;
for each $Q \in H^c_{curr_quorum_group}$:
$p^c(Q) = 1/|H^c_{curr_quorum_group}|$;
for each $Q \in \mathcal{Q} \setminus H^c_{curr_quorum_group}$:
$p^c(Q) = 0$;
WHEN RECEIVING REPLY FOR REQUEST r:
$history[last_req] = response_time(r)$;
$last_req = (last_req + 1) \bmod history_count$;
$avg_rt = (\sum_{i=0}^{history_count} history[i])/history_count$;

Figure 5. DelayBins algorithm at client c



(a) Response time (ms) (b) Throughput (req/s)

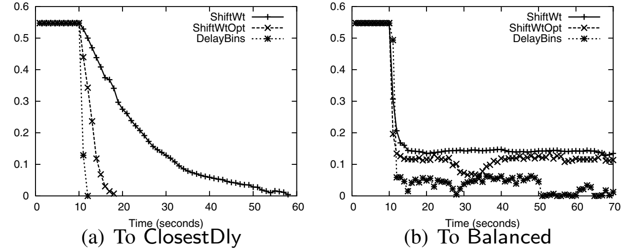
Figure 6. Comparison on Grid



(a) Response time (ms) (b) Throughput (req/s)

Figure 7. Comparison on Majority

We compare the performance of DelayBins to that of ClosestDly and Balanced for different client demands. Figures 6 and 7 show the response time and throughput of DelayBins on the 5×5 Grid, and the $(4b + 1, 5b + 1)$ Majority with $b = 2$, respectively. In both low-load and high-load cases, DelayBins yields access strategies that achieve performance similar to the best access strategies for that load.



(a) To ClosestDly (b) To Balanced

Figure 8. Convergence (L2 Norm) on Grid

To compare the convergence time of DelayBins to that of ShiftWt and ShiftWtOpt, we repeat the following experiments: we start clients with ClosestDly as the initial access strategy (by setting $curr_quorum_group = 1$, i.e., clients use the closest quorums) in a regime of high load (150 clients, or clientsPerNode $= 15$) and measure the time it takes them to change the average of their access strategies to Balanced. We also perform the converse experiment: we start clients from Balanced (i.e., by making clients use the highest group number) and measure how fast their average access strategy converges to ClosestDly with only 10 clients (clientsPerNode $= 1$). We use a value of $history_count = 3$ for our experiments in order to reconcile the contradictory goals of having a quickly adapting algorithm and one that does not react to sporadic spikes in load [37]. Figure 8(a) shows that DelayBins converges much faster than ShiftWtOpt to ClosestDly (in about 1–2 seconds). At the same time, the convergence time to Balanced is about the same as that of ShiftWtOpt.

k	Conv. to ClosestDly		Conv. to Balanced	
	ShiftWtOpt	DelayBins	ShiftWtOpt	DelayBins
2	5 s	1 s	3 s	2 s
3	6 s	1 s	1 s	1 s
4	7 s	2 s	2 s	3 s
5	8 s	2 s	2 s	2 s

Table II
CONVERGENCE TIMES ON $k \times k$ GRID

To perform a more exhaustive comparison, we evaluate the convergence time of DelayBins for various universe sizes. In all cases, DelayBins converges roughly as fast or faster than ShiftWtOpt, as can be seen in Table II. The best explanation for this difference is that ShiftWtOpt bases its decisions on more fine-grained (i.e., per server), but consequently more stale, response-time information.

The last measure with respect to which we evaluate DelayBins is client overhead. Table I lists results for both

DelayBins and ShiftWtOpt. In all cases the overhead of DelayBins is smaller than that of ShiftWtOpt and also similar to that of ClosestDly and Balanced, both of which have an overhead between 1 and 2 μs in our evaluations.

The primary reason for the superior client-side overhead of DelayBins is given by the following observation. Even though the pseudocode of Figure 5 updates the access strategy p^c per quorum each time it is updated (a computational cost proportional to m), it is possible to dramatically optimize this implementation. Specifically, it is not necessary to maintain p^c explicitly; rather, a quorum from $H^c_{curr_quorum_group}$ can be chosen uniformly at random to perform a request, after $curr_quorum_group$ has been updated. The execution time of DelayBins depends primarily on the number of groups G^c_i into which we partition the quorums, which by construction is never greater than the number of servers. Furthermore, these groups do not change often—in practice they need only change when network delays to servers change significantly—and thus can be precomputed. This makes the performance of DelayBins typically depend only on the number of servers.

This is a significant difference from ShiftWtOpt, for which the client overhead grows quickly as the number of quorums grows. In some cases, ShiftWtOpt must go through exponentially many (in the number of servers) quorums to update the access strategy according to which it samples quorums. Even if this is decoupled from the actual process of choosing a quorum, the overhead associated with updating the access strategy can delay the sampling task given large quorum system sizes.

V. State Transfer

Our dynamic quorum selection algorithms as presented in Section IV assume that response time is the only factor that needs to be considered when choosing quorums. However, in many protocols, to switch quorums results in additional cost arising from the need to transfer *state* from servers in the old quorum to servers only in the new quorum. This is of particular concern on a wide area network where bandwidth is limited and delays are relatively large. Here, we discuss the following questions in the context of our algorithms from Section IV: (i) if state transfer is required, does Balanced still perform better than ClosestDly when there is high load; and (ii) if not, what can we do about it?

To make our discussion more concrete, we focus on state-machine-replication protocols [8], which are an important class of protocol that require state transfer. In these protocols, the state of the service is replicated to different machines. The Q/U protocol [1] introduced the concept of a purely quorum-based state-machine-replication protocol, but protocols such as BFT [9], HQ [2], and Zyzzyva [10] also use quorum system properties to maintain state. Because updates to the state are generally incremental and based on the previous state, a replica typically needs a copy of the current state in order to perform such an operation. As this state can be modified at a quorum of servers, some servers may not always have the current state. Transferring the state to them is where the costs of state transfer can become an issue.

Q/U uses a type of majority quorum system called opaque [25] in which most servers are in every quorum and therefore have the current state as described in Section IV-A. Even so, the Q/U authors have found the cost of state transfer to be relatively large even on a switched local area network where a peak throughput reduction of roughly 40% was observed [1] (see below). To avoid this cost, the authors introduce the notion of a *preferred quorum* [1]. In Q/U, the state is divided into *objects*. Each object is accessed at its preferred quorum by default. Since Q/U was originally evaluated on local area networks with homogeneous, low latencies, the preferred quorums are spread uniformly across the quorum system so as to balance load. The reduction in throughput mentioned above occurs when objects are accessed at quorums chosen uniformly at random—thereby requiring state-transfer—instead of at their preferred quorums.

Using the notion of preferred quorums, we propose the following algorithm called Sticky. Whereas Balanced and ClosestDly are per-client access strategies, Sticky is, in a sense, a per-object access strategy. In Sticky, each object is accessed at its preferred quorum by default, and preferred quorums are placed so as to balance the load on each server given the anticipated workload. More specifically, Sticky is defined by a mapping $\mathsf{pref}()$ where $\mathsf{pref}(o)$ is the preferred quorum for object o. A workload induces a distribution π on objects (i.e., $\sum_o \pi(o) = 1$), in the sense that o is accessed in $\pi(o)$ fraction of the accesses. Therefore, the load on a server s, given a mapping $\mathsf{pref}()$ and workload π is,

$$\mathsf{load}(s) = \sum_{o:s\in\mathsf{pref}(o)} \pi(o).$$

For any such anticipated distribution π on object accesses, Sticky is defined by a mapping $\mathsf{pref}()$ that minimizes the imbalance between the most loaded and least loaded servers, i.e.,

$$\arg\min_{\mathsf{pref}()} \max_{s,s'} |\mathsf{load}(s) - \mathsf{load}(s')|.$$

To evaluate this, we run a version of Q/U that we have updated to facilitate wide-area experiments (available as Q/U v1.3 [40]). This version of Q/U allows us to create a much larger number of clients using a small number of client machines. Since Q/U is currently of practical interest for small numbers of servers, we emulate a hypothetical six-server topology based on round-trip times in the USA. We configure a new Modelnet topology with data from Table III, which is based closely on network delay measurements derived from AT&T (http://ipnetwork.bgtmo.ip.att.net). We

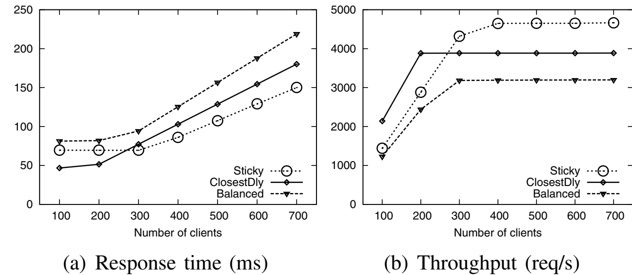
deploy this topology on the rack of machines described in Section IV-A.

As seen in Table III, our topology consists of six network sites (San Francisco, Denver, Chicago, Cleveland, New York City, and Cambridge) located across the USA with round-trip latencies between sites as given in the table. Each site hosts one server. All clients are connected to Cambridge with zero latency and no specified bandwidth restrictions. There is one server per site, and each server is connected to its site by two unidirectional links each with 15Mbps bandwidth and zero latency. This bandwidth restriction decreases the maximum throughput of Q/U from approximately 25,000 req/s on our hardware to about 5000 req/s, as seen below, bringing the amount of data sent per second into a range that Modelnet (which must forward all of the network traffic for both clients and servers) can support on our hardware. Given that this bandwidth allocation is for a single service (i.e., not for the entire site), we feel it is reasonable. Furthermore, based on experimentation with other bandwidth levels, we expect that our results generalize. Because each client is situated in Cambridge, the closest quorum is the one that omits the server in San Francisco. This quorum requires at least 46 milliseconds round-trip latency to access. Any quorum that contains the server in San Francisco requires at least 76 milliseconds latency.

	den	**chi**	**cle**	**nyc**	**cam**
sfo	30	50	56	70	76
	den	20	26	40	46
		chi	6	20	26
			cle	14	20
				nyc	6
					cam

Table III
ROUND TRIP NETWORK LATENCY DATA (MS)

In our experiments, each client is a thread with exclusive access to its own object maintained by the service. Each client submits a request (of trivial size) to modify the object and waits for a response. Upon receiving a response that the object has been updated, the client immediately issues another modify request. As such, the workload is uniform. (Later, we address issues that may arise from other workloads.) Because clients wait to issue requests, requests are issued at the rate at which responses are received [41]. It takes at least the network round-trip latency for each client to receive a response, and so, when the network latency is longer, more clients are required to drive the service to its peak capacity. For the purposes of our evaluation, low load can be thought of as any number of clients insufficient to drive the service to its peak capacity, while high load as any number of clients achieving peak capacity. If the client has selected a different quorum for the next request, the state of the object must be transferred to the server that is only in the new quorum before that server can update the object. Note that, although nonzero, the processing cost per request at each server is much less than the 1 ms cost induced in Section IV. This explains the higher throughput observed in this set of experiments compared to those in Section IV.

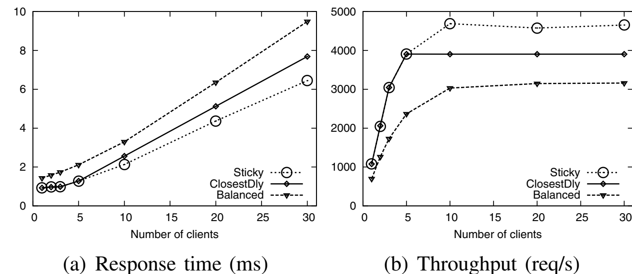


(a) Response time (ms) (b) Throughput (req/s)

Figure 9. Q/U with state transfer

Because of state transfer, as Figure 9 shows, ClosestDly indeed outperforms Balanced given this network configuration. However, in the presence of state transfer, Sticky behaves much as Balanced does with no state transfer. In particular, Sticky outperforms ClosestDly in periods of high load, while ClosestDly performs better than Sticky when there is low load.

To understand the impact of network latency, we repeat the above analysis, but with zero latency between sites (instead of the values from Table III). The results are given in Figure 10. As expected, because all quorums are equally close, Sticky and ClosestDly perform equally well here when there is low load. However, Sticky again outperforms ClosestDly when there is high load. In all cases, as before, Balanced performs worse than either Sticky or ClosestDly because of its need for state transfer.



(a) Response time (ms) (b) Throughput (req/s)

Figure 10. Q/U with state transfer, no delay

The use of Sticky as a substitute for Balanced in the algorithms of Section IV requires a way to move preferred quorums. A sketch of a conceptually simple modification to Q/U in order to do so is as follows. First, note that we can use Q/U itself to maintain the location of the preferred quorum for each object. Therefore, to update the preferred quorum, a client performs an update operation specifying the new preferred quorum. This operation for updating an object's preferred quorum is performed like any other (linearizable) operation for the service, so that it is unambiguous

as to what the preferred quorum is at any logical point in time. When the object is updated, the operation can be specified as *contingent* on the selected quorum being the preferred quorum (to maintain efficiency by avoiding state transfer), or as *without contingencies* (which might cause state transfer). In the former case, servers reject the operation if the selected quorum is not the preferred quorum for the accessed object. They do this on the basis of state information, included with the operation, that is relayed by clients from servers to servers (called object history sets [1] in Q/U terminology) for the preferred quorum. In such a case, the client must resubmit the operation.

To learn the preferred quorum, a client uses a query operation specifying the object. Although the client can use a query to lookup the preferred quorum for an object, optimizations such as the following minimize such lookups (so as to minimize the number of rounds of interaction on the critical path of an operation). A client caches the preferred quorum following each lookup, and performs its subsequent operations using contingent requests. Upon servers rejecting this operation due to the preferred quorum changing, the client can be informed in the rejection of the new preferred quorum, or it can lookup the preferred quorum again. The client issues requests without contingencies when the preferred quorum is unresponsive.

Sticky maximizes throughput and minimizes average response time when there is high load (as does Balanced when there is no state transfer). Despite this, if multiple clients access the same object, the placement of a given preferred quorum can be better in terms of individual response times for certain clients than for others (e.g., when some clients are close to the preferred quorum while others are far from it). This may give incentive to clients to move preferred quorums close to themselves. However, the benefits of Sticky compared with Balanced can be negated without well-defined policies for updating preferred quorums. For example, a client that changes the preferred quorum to be closer to itself may cause state transfer, both now and again later if another client moves the preferred quorum back. It remains an open question to develop policies for changing preferred quorums for any particular workload.

The take-away message from the analysis in this section is that state transfer can have an impact on the performance of any algorithm that switches quorums. In such situations, algorithms such as Sticky that balance load but avoid state transfer should be considered as a substitute for Balanced in the algorithms of Section IV. However, for systems that do not require state transfer, the algorithms of Section IV can be employed directly.

VI. Conclusion

In many wide-area settings, a constant, unchanging workload is not a certainty. Unpredictable events, such as flash crowds caused by nearly instantaneous popularity of services, can cause servers that are expected to respond quickly to instead suddenly respond slowly. In this paper, we have shown how clients can change their quorum selections efficiently and in isolation in order to react to such changes. This allows them to achieve good performance, not only for themselves, but in a way that causes the overall system performance to converge rapidly to that of the best global strategy for the new workload. Our low-overhead algorithms presented in Section IV are suitable for allowing clients to change their quorum selections based on limited knowledge. These algorithms can react to changing workloads while producing solutions that lead to very good system performance.

One important use of quorum systems is in the maintenance of state. In such systems, a switch to a new quorum may result in the need to transfer the state from the old quorum to the new quorum. On a wide area network, this can be an expensive operation. As such, in Section V, we have proposed and evaluated an alternative way to balance the load across the servers when state transfer is a concern.

Acknowledgment

This work was supported in part by NSF grants CCF-0424422 and CNS-0756998.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Symposium on Operating Systems Principles*, 2005.

[2] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[3] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind, "Persistent objects in the Fleet system," in *Information Survivability Conference and Exposition*, vol. 2, Jun. 2001, pp. 126–136.

[4] F. Oprea and M. K. Reiter, "Minimizing response time for quorum-system protocols over wide-area networks," in *International Conference on Dependable Systems and Networks*, June 2007.

[5] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: characterization and implications for cdns and web sites," in *International Conference on World Wide Web*, 2002, pp. 293–304.

[6] V. N. Padmanabhan and K. Sripanidkulchai, "The case for cooperative networking," in *International Workshop on Peer-to-Peer Systems*, 2002, pp. 178–190.

[7] S. Ranjan, J. Rolia, E. Knightly, and H. Fu, "Qos-driven server migration for internet data centers," in *International Workshop on Quality of Service*, 2002.

[8] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[9] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002.

[10] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," in *Symposium on Operating Systems Principles*, 2007, pp. 45–58.

[11] A. W. Fu, "Delay-optimal quorum consensus for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 1, pp. 59–69, 1997.

[12] D. Golovin, A. Gupta, B. Maggs, F. Oprea, and M. Reiter, "Quorum placement in networks: Minimizing network congestion," in *Symposium on Principles of Distributed Computing*, 2006.

[13] A. Gupta, B. Maggs, F. Oprea, and M. K. Reiter, "Quorum placement in networks to minimize delays," in *Symposium on Principles of Distributed Computing*, 2005.

[14] N. Kobayashi, T. Tsuchiya, and T. Kikuno, "Minimizing the mean delay of quorum-based mutual exclusion schemes," *Journal of Systems and Software*, vol. 58, no. 1, pp. 1–9, 2001.

[15] X. Lin, "Delay optimizations in quorum consensus," in *International Symposium on Algorithms and Computation*, 2001, pp. 575–586.

[16] T. Tsuchiya, M. Yamaguchi, and T. Kikuno, "Minimizing the maximum delay for reaching consensus in quorum-based mutual exclusion schemes," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 4, pp. 337–345, 1999.

[17] L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar, "Dual-quorum replication for edge services," in *Middleware*, Dec. 2005, pp. 184–204.

[18] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling Byzantine fault-tolerant replication to wide area networks," in *International Conference on Dependable Systems and Networks*, Jun. 2006, pp. 105–114.

[19] R. L. Carter and M. E. Crovella, "Server selection using dynamic path characterization in wide-area networks," in *INFOCOM*, 1997.

[20] J. D. Guyton and M. F. Schwartz, "Locating nearby copies of replicated internet servers," *SIGCOMM Comput. Commun. Rev.*, vol. 25, no. 4, pp. 288–298, 1995.

[21] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Topologically-aware overlay construction and server selection," in *INFOCOM*, 2002.

[22] D. Starobinski and T. Wu, "Performance of server selection algorithms for content replication networks," in *NETWORKING*, 2005, pp. 443–454.

[23] S. Ranjan, R. Karrer, and E. W. Knightly, "Wide area redirection of dynamic content by internet data centers," in *INFOCOM*, 2004.

[24] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM Journal on Computing*, vol. 27, no. 2, pp. 423–447, 1998.

[25] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[26] D. Malkhi, M. K. Reiter, and A. Wool, "The load and availability of Byzantine quorum systems," *SIAM Journal on Computing*, vol. 29, no. 6, pp. 1889–1906, 2000.

[27] R. Holzman, Y. Marcus, and D. Peleg, "Load balancing in quorum systems," *SIAM J. Discret. Math.*, vol. 10, no. 2, pp. 223–245, 1997.

[28] Y. Amir and A. Wool, "Evaluating quorum systems over the Internet," in *Symposium on Fault-Tolerant Computing*, Jun. 1996.

[29] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

[30] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: Building efficient replicated state machine for WANs," in *Symposium on Operating Systems Design and Implementation*, 2008.

[31] K. P. Gummadi, S. Saroiu, and S. D. Gribble, "King: estimating latency between arbitrary internet end hosts," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, 2002.

[32] T. S. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *INFOCOM*, 2002.

[33] S. Y. Cheung, M. H. Ammar, and M. Ahamad, "The grid protocol: A high performance scheme for maintaining replicated data," *Knowledge and Data Engineering*, vol. 4, no. 6, pp. 582–592, 1992.

[34] A. Kumar, M. Rabinovich, and R. K. Sinha, "A performance study of general grid structures for replicated data," in *International Conference on Distributed Computing Systems*, 1993, pp. 178–185.

[35] D. K. Gifford, "Weighted voting for replicated data," in *Symposium on Operating Systems Principles*, 1979, pp. 150–162.

[36] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180–209, 1979.

[37] F. Oprea, "Quorum placement on wide-area networks," Ph.D. Thesis, ECE Department, Carnegie Mellon University, 2008.

[38] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak, "Operating system support for planetary-scale network services," in *Symposium on Networked Systems Design and Implementation*, Mar. 2004.

[39] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 271–284, 2002.

[40] http://www.pdl.cmu.edu/QU/index.html.

[41] G. Banga and P. Druschel, "Measuring the capacity of a Web server under realistic loads," *World Wide Web*, vol. 2, no. 1-2, pp. 69–83, 1999.