Quorum Placement on Wide-Area Networks

Florian Oprea

May 2008

Dept. of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:

Prof. Anupam Gupta (Carnegie Mellon University)
Prof. Bruce M. Maggs (Carnegie Mellon University)
Prof. Michael K. Reiter (University of North Carolina at Chapel Hill) , Chair
Dr. Lidong Zhou (Microsoft Research Silicon Valley)

© 2008 Florian Oprea

Abstract

Content distribution networks are the dominant technology for distributing shared media on todays Internet. There are several types of content for which these systems have been proven highly successful: static databases, streaming media, online gaming. At the same time this architecture is not appropriate for other types of applications such as transactional databases that require both strong consistency and wide area replication. One scalable technique that can cope with these requirements is to deploy quorum systems on wide area networks. The most important drawback of this approach comes from the performance penalties that one has to pay when accessing data over the wide area.

This thesis makes several steps towards addressing this performance problem. First, it presents approximation algorithms for deploying quorum systems to approximately minimize network delay or network congestion. Second, it presents several heuristics for balancing the impact server load and network delay have on client response times and overall throughput. Third, it shows that the techniques presented here can be effectively used to improve performance through both simulation and emulation of real quorum systems protocols on several wide area network topologies.

Acknowledgments

I would like to thank my advisor Mike Reiter for his constant support over the years. The completion of this thesis owes a great deal to his efforts. His contribution extends far beyond the current work though. He invested a considerable amount of time in trying to make me a better researcher: by giving me technical advice, by working on my presentation skills, by pulling me out of dead ends I was stuck in and suggesting different approaches or new research directions altogether. Besides nurturing my interests for problems in distributed systems he has also spent much time teaching me core concepts in network security and applied cryptography, two other fields that I am very fond of. Thanks Mike for all of this and much more!

I would akso like to thank the three other people that had the most influence on me during my doctorate studies. Anupam and Bruce for the many hours spent discussing on quorum systems problems and explaining many complex concepts in approximation and randomized algorithms. Lidong who has intoduced me to the Paxos protocol and to other very interesting distributed systems problems, during my summer internship at MSR Sillicon Valley.

Many other people provided either moral or material support to the realization of this thesis. Of these I would like to bring special thanks to Michael Abd-El Malek who helped me run experiments with the Q/U protocol on srs and provided extensive support during the many times when things would not work as expected. I would also like to thank Mike Merideth for dilligently reviewing a draft of this thesis. My life in Pittsburgh would have been much duller without friends like Asad, Charles and Scott who were always there when I needed them.

Finally I would like to thank my family for all their support and encouragements over these 6+ years, particularly my wife Alina for her patience during the last few months and my mother-in-law Maria and my mom for taking care of Andrei during the time I spent working on this thesis. Last but not least I would like to thank both of my parents, Dumitru and Mariana, who over the years invested so much personal time into my education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Content distribution networks like Akamai are a dominant technology for distributing highly popular content on the Internet today. The success of these systems relies on several mechanisms: replication, web caching, request redirection and load balancing, to name the most important. These techniques increase service availability and bring content closer to end-users, thereby minimizing response time and bandwidth consumption in the core of the network. While in general these techniques are sufficient for distributing static content or content that does not change too frequently, they are not sufficient for applications where the service state changes at a fast rate. Furthermore, some of these applications, such as electronic stock exchange services or online gaming, also require strong consistency semantics (e.g., linearizability - Herlihy and Wing [1990]) and need the same performance guarantees as the applications based on static content. Most proposals put forth to address these concerns (e.g., Sivasubramanian et al. [2005]) submit updates to a single node, but this creates a bottleneck and a single point of failure in the system. An alternative that we explore in this thesis is the use of *quorum systems*.

Quorum systems were originally proposed in the database community (Gifford [1979], Thomas [1979]) more than 25 years ago. A *quorum system* is a collection of subsets (*quorums*) of a universe U 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 the servers of some quorum (possibly in more than one round). The intersection property enables a read or update to observe the effects of prior updates, while the fact that not all servers need to be contacted for each operation enables increases in efficiency by dispersing load across servers.

When using quorums in wide-area applications, a number of factors affect the response

times realized by clients. First, the location of servers, or more to the point, the proximity of servers to clients, is an important factor in minimizing response times, since if all quorums contain far-away servers, then the network costs for each request will be substantial. Second, when a client selects a quorum to use in an operation—which it does via its *access strategy*—a tension can arise between the goals of dispersing load across servers and minimizing network costs for accessing quorums. On the one hand, network costs can typically be reduced by clients accessing quorums closest to them. On the other hand, processing delays may be best dispersed among the servers by clients utilizing quorums that are less heavily loaded but that might be further away.

This thesis provides solutions to address these issues for various notions of network and processing costs. The first two chapters develop algorithms to *place* the servers U on the nodes of a physical network so as to respect nodes' processing constraints but otherwise minimize the network delay of (Chapter 2), or the network congestion induced by (Chapter 3), client accesses. Chapter 4 provides an empirical evaluation of the results in Chapter 2, and further extends them with algorithms to refine placements and optimizing clients' access strategies to minimize client response times based on the client demand anticipated for the service. Finally, Chapter 5 presents algorithms by which clients can dynamically adapt their access strategies when workloads change in order to keep their response times low.

1.1 Minimizing network delays

The problem of minimizing network delays for accessing quorums has been considered before (e.g., Fu [1997], Tsuchiya et al. [1999], Kobayashi et al. [2001], Lin [2001]), but these works do not consider the processing load on the servers. Here we use a different approach: we develop a method for mapping an arbitrary quorum system onto a network in a way that approximately minimizes network delay and does not incur too much load on any network node at the same time. The idea here is that one can then choose a specific quorum construction that is optimal with respect to some measure and map it on a network in a way that provides "good" network delay at the same time.

More precisely, the objective function we seek to minimize is the average network delay observed by clients when accessing the quorum system, in one of two senses. The first applies to the case when clients access the servers of a quorum in parallel. In this case the latency incurred is given by the maximum distance from the client to any of the quorum members. We call this cost function *max-delay*. The second function applies to the case when quorum members are accessed sequentially. We call this function *total-delay*.

For each of these two cost functions we study what we call the *quorum placement problem*: the problem of placing a quorum system (or rather its universe of servers) on a network in a way that minimizes network delay and does not exceed node capacities at the same time. In Chapter 2 we show that the Quorum Placement Problem for the max-delay cost function for arbitrary quorum systems and arbitrary networks is NP-hard. We can, however, get to within a factor of $5 * \alpha/(\alpha - 1)$ from the optimal solution if we are allowed to exceed the node capacities by a factor of α. Also for two specific quorum constructions, notably Majority (Gifford [1979], Thomas [1979]) and Grid (Cheung et al. [1992], Kumar et al. [1993]), we can decrease the approximation factor to 5. For those cases we also present simpler placement algorithms.

To obtain the first approximation result for the max-delay cost function, we use the following technique. We first reduce the original problem to the case with a single client (a problem we call Single-Source Quorum Placement Problem) losing a factor of 5 in the process. We then show that Single-Source Quorum Placement Problem is NP-hard but can be approximated to within a factor of $\alpha/(\alpha-1)$ from the optimal solution if we are allowed to exceed the node capacities by a factor of α. To obtain this result we write Single-Source Quorum Placement Problem as an integer program and solve its linear relaxation to obtain a fractional solution x. We then apply a filtering step to x to ensure that no universe element is fractionally assigned to nodes too far away from the single client. Finally we round this fractional solution to an integral one using results for the *generalized assignment problem* (GAP) due to Shmoys and Tardos [1993].

For the total-delay cost function we show exact placement algorithms, provided we are allowed to exceed node capacities by a factor of two. To obtain these results we use similar techniques: we first express the problem as an integer program, solve its linear relaxation to obtain a fractional solution and then round the fractional solution using GAP to obtain an integral solution with the same cost as the optimal (but doubling node capacities in the process).

1.2 Minimizing network congestion

The next problem we study is that of minimizing network congestion induced by client accesses to a quorum system. We use a similar setup: we study the problem for arbitrary quorum systems and for arbitrary networks. This is also a quorum placement problem: we want to map a quorum system (or, rather, its universe U) onto a network in a way that approximately minimizes network congestion and does not exceed the network nodes' capacities.

We consider two routing models for this problem. In the first, packets can travel between two hosts over any path between the two, the choice of which path to use for a particular packet being made probabilistically. This model is particularly suited for overlay networks where the path used by a particular packet can be chosen at the application level. We call this model *the arbitrary routing* model. The probabilities used to select paths for each packet are chosen such that the maximum expected congestion on any edge is "close" to optimal. More precisely, in this model we show that the Quorum Placement Problem for Congestion is NP-hard, but can be approximated to within a polylog factor of the size of the network provided we are allowed to double the capacity of each node. This result holds for arbitrary graphs. For trees we decrease the approximation factor for congestion to 5.

In the second model we consider that paths between hosts are fixed in advance. This is the model commonly used at the IP level in today's Internet infrastructure. We call this the *fixed paths* model. We show that in the fixed paths model Quorum Placement Problem for Congestion is also NP-hard, but can be approximated if we double the capacity of each network node. The exact approximation ratio is $O(\frac{\eta \cdot \log n}{\log \log n})$, where n is the size of the network (number of nodes), $\eta \leq |U|$ and $\eta = 1$ if the access strategies employed by clients balance load across servers equally.

The results for the general case of Quorum Placement Problem for Congestion in the arbitrary paths model are obtained by reducing the problem to the case in which the graph is a tree *and there is only one client in the system*. This reduction uses some of the properties of quorum systems, combined with a general graph decomposition result of Racke [2002]; however, it costs us a factor of $O(\log^2 n \log \log n)$ in the congestion. For the single-client tree case, we give an approximation algorithm by first writing an integer programming formulation, and then rounding its linear-programming relaxation: the rounding uses an algorithm for unsplittable flows from Dinitz et al. [1999].

The results for the fixed paths model use a different set of tools: we first develop an algorithm for instances where all the elements of U have identical ("uniform") loads. For this we use a different linear programming relaxation, and then round it using a different rounding technique that does not allow node capacities to be violated (Srinivasan [2001]). We then use this algorithm as a subroutine to solve the non-uniform case by carefully placing sets of elements in decreasing order of their loads.

We conclude Chapter 3 with a set of results showing that migration of universe elements between physical nodes of the network can help further reduce congestion.

1.3 Minimizing response time for static workloads

In Chapter 4 we empirically evaluate the quorum placement algorithms of Chapter 2 in a realistic setting. The topologies emulated in our evaluation are obtained from two sources: from ping round-trip time measurements between different PlanetLab sites (Bavier et al. [2004]) and from one-way latency measurements between DNS nameservers performed with a DNS-based tool (Gummadi et al. [2002]). In addition, we extend these algorithms to refine client access strategies to minimize response times based on the anticipated service load.

Specifically we assume that the response time observed by a client when contacting a server is the sum of the network latency to reach that server and the amount of time the client's request waits to be serviced at the server. We use a simple estimator for this waiting time, namely that it is proportional to the fraction of requests that target that server. We show that in the case of low client demand, response time is well approximated by network delay and so the best access strategy for clients in this case is to always contact the closest quorum. We call this access strategy ClosestDly. Our evaluation of the placement algorithms from Chapter 2 for the Majority and Grid quorum systems shows that, for some topologies, network delay increases slowly with the universe size up to a certain point and then increases very fast until it reaches a saturation point. Another important observation is that up to fairly large universe sizes, network delay is not much worse than that of a single server, which implies that in this case increased fault-tolerance can be achieved at a relatively low cost.

In the case of extremely high client demand, load at servers offsets the impact of the network delay in client response time. In this case an access strategy that balances load across servers is the best one. We call this access strategy Balanced.

We also show that there is a range of client demand values where none of ClosestDly and Balanced yields the best client response times. To find the best access strategy in this case we propose two heuristics that work as follows. We start with an LP-formulation of the problem having as variables the clients' access strategies and as objective function the average network delay observed by clients. For a fixed set of node capacities, we solve the LP to get the optimal solution, that is the set of access strategies minimizing network delay, and then compute the response time for that set of access strategies. By varying the network nodes' capacities we can thus do a search for the optimum set of access strategies (i.e., minimizing response time) in this space. The two heuristics differ only in the way they set node capacities. Our evaluation shows that these heuristics give better results than the ClosestDly and Balanced access strategies for this range of client demand.

1.4 Minimizing response time for dynamic workloads

In Chapter 5 we consider the problem of adapting the clients access strategies in response to changes in workload. To this end we propose two quorum selection algorithms for dynamic workloads. The algorithms enable each client to independently choose the quorum it accesses for each operation it invokes. The client does so with no communication with other clients and with only the information it gleans from previous operation invocations to the service. As such, these algorithms impose negligible overheads on servers. We show that our algorithms quickly adapt clients' choices of quorums in the face of workload dynamics, and that the clients' quorum choices that result minimize client response times and maximize client throughput nearly optimally across a broad range of workloads. We draw these conclusions based upon evaluations of our algorithms on two wide-area, emulated topologies derived from PlanetLab measurements and using two quorum-based operation-invocation protocols: a generic Grid quorum (Cheung et al. [1992], Kumar et al. [1993]) protocol and the Q/U (Abd-El-Malek et al. [2005]) protocol using a Majority quorum system.

The first algorithm is based on the following simple weight shifting strategy: a client monitors the response time it gets from servers across service invocations and sorts the quorums according to their response time (the response time of a quorum being the maximum response time of any of its members). After each request the client moves half of the weight from a quorum with higher response time to a quorum with a lower response time. Our evaluation shows that this algorithm is able to adapt to changing workloads but the time it takes to do so is fairly large (on the order of tens of seconds). We suggest several optimizations that shift more weight in each time step and consequently adapt much faster to changing workloads.

One problem with the weight-shifting algorithm is that it does not scale for quorum systems with a very large number of quorums. To address this problem we propose a second algorithm whose running time depends only on the number of servers. Here we partition the quorum system according to the network delay from the client to each quorum. This creates a number of groups no larger than the number of servers. For each request the client then chooses the quorum to access by sampling uniformly at random from a certain group. To choose the group used for sampling the client monitors the response time of the past requests and starts including more distant groups as response time increases above a certain threshold or starts dropping the most distant group as response times decrease below a certain threshold.

Our evaluation shows that the two algorithms quickly adapt the clients' access strategies in response to wokload dynamics, while imposing only small overheads to clients.

Chapter 2

Quorum Placement in Networks: Minimizing Network Delays

In this chapter we address the problem of placing quorum systems on wide-area networks so as to minimize two network delay objective functions: the average max-delay and the average total-delay. We start with some background information on quorum systems and continue with a description of the model. We then formalize each problem considered and briefly describe our main results along with the techniques used in each case. We present in more detail our results for the max-delay-based objective function in Sections 2.2 and 2.3 and the results for the total-delay-based objective function in Section 2.4.

2.1 Background and model

Given a *universe* U of elements, a *quorum system* $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ on U is a family of subsets of U such that any two quorums Q_i and Q_j have non-empty intersection. When using quorum systems a client needs to issue a request to only a quorum Q_i of servers. The quorum used in an access is usually selected according to a probability distribution $p : \mathcal{Q} \rightarrow [0, 1]$ over $\mathcal{Q}$; this is called an *access strategy* for the quorum system. Each access strategy p induces a *load* on each element $u \in U$ defined as $\mathsf{load}_p(u) = \sum_{Q_i \ni u} p(Q_i)$. The load of a quorum system corresponding to the access strategy p is defined as $\mathsf{load}_p(\mathcal{Q}) = \max_{u \in U} \mathsf{load}_p(u)$, i.e., the maximum load of any element $u \in U$. Although one could use any access strategy for sampling a quorum, in practice people use the access strategy minimizing the load of the quorum system. In the past much research has been devoted to developping quorum systems $\mathcal{Q}$ and associated access strategies p that minimize load, see

e.g., Naor and Wool [1998].

We model the underlying network as an undirected graph $G = (V, E)$, of size $|V| = n$, with each node having an associated capacity $\mathsf{node\_cap}(v) \in \mathbb{R}^+$ to handle requests. There is a positive "length" $\mathsf{length}(e)$ for each edge $e \in E$, which induces a distance function $d : V \times V \to \mathbb{R}^+$ obtained by setting $d(v, v')$ to be the sum of lengths of the edges comprising the path from v to v' that minimizes this sum (i.e., the shortest path). We assume that the set of clients wanting to make quorum accesses is V.

Our goal is to determine a map $f : U \to V$ (which we call a *placement* of the quorum $\mathcal{Q}$ on the nodes of G) that preserves the load of the quorum system (in a sense to be defined) and produces good network delay at the same time. We define the load of any network node $v \in V$ as $\mathsf{load}_f(v) = \sum_{u \in U : f(u) = v} \mathsf{load}(u)$. Ideally the load of any node should not exceed its capacity $\mathsf{node\_cap}(v)$.

The first notion of network delay we deal with is called *average max-delay* and is motivated by the following usage scenario. Given a placement f, if a client $v \in V$ accesses a quorum Q by contacting servers in parallel, then the time required to reach all elements of quorum Q is proportional to the *maximum* distance from v to any member of Q. Hence we model the *delay* as the distance of v to the farthest-away element of Q:

$$\delta_f(v, Q) = \max_{u \in Q} d(v, f(u)). \tag{2.1.1}$$

(We call this the *max-delay* access cost.) Then, the *expected delay* (under p) for v to access $\mathcal{Q}$ is

$$\Delta_f(v) = \sum_{Q \in \mathcal{Q}} p(Q)\, \delta_f(v, Q). \tag{2.1.2}$$

Note that if each client $v \in V$ makes quorum accesses at the same rate, then the average delay of the entire system will be $\mathsf{Avg}_{v \in V}[\Delta_f(v)] = \frac{1}{n} \sum_v \Delta_f(v)$. (For ease of exposition, we focus on this uniform-rate case; we can extend our results to more general cases when different clients make quorum accesses at different rates.)

We are finally in a position to formally define the first problem studied in this chapter:

Problem 2.1.1. (Quorum Placement Problem (QPP)) *Given a quorum system $\mathcal{Q}$ over the universe U, along with an access strategy p, and also an undirected network $G = (V, E)$ with capacities $\mathsf{node\_cap} : V \to \mathbb{R}^+$ and inter-vertex distances $d(\cdot, \cdot)$, find a placement $f : U \to V$ that (a) minimizes $\mathsf{Avg}_{v \in V}[\Delta_f(v)]$ (i.e., "low delay") subject to (b) $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$ for all $v \in V$ (i.e., "low load").*

Our main result for this problem is the following:

Theorem 2.1.2. *Let f^* be an optimal solution to the Quorum Placement Problem. Then, for any $\alpha > 1$, we can find in polynomial time a placement f with $\mathsf{load}_f(v) \leq (\alpha + 1)\,\mathsf{node\_cap}(v)$ for all $v \in V$, and for which*

$$\mathsf{Avg}_{v\in V}[\Delta_f(v)] \quad \leq \quad \tfrac{5\alpha}{\alpha-1}\,\mathsf{Avg}_{v\in V}[\Delta_{f^*}(v)] \tag{2.1.3}$$

Hence, e.g., we can find a map f that exceeds the capacities on the nodes by a factor of 4, but ensures that the delay is within a factor of 7.5 of the optimal delay of f^*.

We then go on to consider some well-known quorum systems, and show the following:

Theorem 2.1.3. *Consider the Grid (Cheung et al. [1992], Kumar et al. [1993]) or Majority (Gifford [1979], Thomas [1979]) quorum systems on U with the access strategy p being the uniform distribution over all quorums. Given any graph G, we can find placements f such that $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$ for each $v \in V$, and where the average max-delay is at most 5 times the optimum among all such solutions.*

We continue with the study of quorum placement for another natural notion of access cost from a client to a quorum, the *total delay*, which captures the delay incurred if quorum elements are contacted sequentially. Specifically, if the vertex v accesses the quorum Q, then let $\gamma_f(v, Q) = \sum_{u\in Q} d(v, f(u))$ be the *total delay* for this access. Given access strategy p, the *expected total delay* for v is $\Gamma_f(v) = \sum_{Q\in\mathcal{Q}} p(Q)\,\gamma_f(v, Q)$. As before, we will be looking for a placement f to minimize $\mathsf{Avg}_{v\in V}[\Gamma_f(v)]$. Our main result for this measure is:

Theorem 2.1.4. *We can find, in polynomial time, a placement $f : U \to V$ where $\mathsf{load}_f(v) \leq 2\,\mathsf{node\_cap}(v)$ at each node $v \in V$ and that has average delay $\mathsf{Avg}_{v\in V}[\Gamma_f(v)]$ at most the optimal delay among all placements satisfying $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$.*

2.1.1 Roadmap and Techniques

Let us give a brief review of the ideas and techniques used to obtain our first result. In Section 2.2, we prove a crucial structural result that guides the section on the max-delay cost function. Specifically, we show that, for any placement f that is a solution to the Quorum Placement Problem for max-delay, there exists a special node v_0, such that even if all the requests are routed through v_0, the average max-delay is at most 5 times that of f. (Hence the additional delay incurred by taking such a detour is not too large, a somewhat surprising fact.) Of course this is only a structural result: no practical algorithm would want to route all the requests through a single node, for fear of creating a bottleneck and a single point of failure.

However, we can now use this result to derive the following important consequence. The average delay of this "relay-via-v_0" routing strategy can be decomposed into two parts: **(i)** the average delay from the clients $v \in V$ to v_0 (which is a constant), and **(ii)** the delay from v_0 to a random quorum of $\mathcal{Q}$ chosen from p, which is just $\Delta_f(v_0)$. Hence, minimizing the overall average delay in this relaying strategy is equivalent to minimizing the average delay for the special case of the Quorum Placement Problem when v_0 is the only client in the system. This allows us to focus, for the rest of Sections 2.2 and 2.3 on this "single-source" version of QPP; we show that any solution to the Single-Source Quorum Placement Problem can be translated back to a solution for Quorum Placement Problem with only a factor 5 loss in the average delay.

In Section 2.2, we formalize the Single-Source Quorum Placement Problem, and show it is NP-hard. We then present an approximation algorithm for it that achieves an average delay of at most $(\frac{\alpha}{\alpha-1})$ times the optimal, if we allow the load on each node to violate the given capacities by a factor of $(\alpha + 1)$. Combining this with our structural lemma (and hence incurring a loss of a factor of 5), we get the algorithm claimed in Theorem 2.1.2.

In Section 2.3, we present optimal solutions for the Single-Source Quorum Placement Problem for two well-known quorum systems, the **Grid** (Cheung et al. [1992], Kumar et al. [1993]) and the **Majority** (Gifford [1979], Thomas [1979]). Combining this with the above reduction, we immediately get Theorem 2.1.3.

In Section 2.4, we address the total delay measure and prove Theorem 2.1.4. Finally, in Section 2.6, we summarize and discuss extensions of our results.

2.2 Maximum delay access cost

In this section we address the Quorum Placement Problem, and give our results for the *max-delay* access cost. Our first result is the following simple yet crucial structural result. Imagine that we are given a quorum placement $f : U \to V$ for the quorum system $\mathcal{Q}$. Then we find a special node v_0, such that routing all the requests to elements in $f(U)$ via the node v_0 causes the average access delay to be ≤ 5 times the delay when we route the requests to $f(U)$ along shortest paths. In other words, even though each message in the system takes a detour via v_0, the average delay does not increase by much, which we feel is a somewhat surprising fact.

Although this result seems to have no practical relevance (because we clearly don't want to route all the traffic through a single node), it gives us a way of accounting for, and approximately minimizing the average delay in the Quorum Placement Problem: we

can split the delay $\Delta_f(v)$ for any vertex v when using this "relay-via-v_0" strategy into two components—the first corresponding to the delay in getting from v to v_0, and the other corresponding to the delay from v_0 to the quorums of $\mathcal{Q}$. But the former contribution, when averaging over all the clients, is a constant. Hence to minimize delay, it suffices to search for placements of $\mathcal{Q}$ that minimize the average delay incurred when the single node v_0 issues all the requests, which is $\Delta_f(v_0)$. To this end, we define the Single-Source Quorum Placement Problem at the end of Section 2.2.1.

We show that minimizing the delay in this new single-client problem is NP-hard for general quorum systems; the proof of this theorem appears in Section 2.2.2. We then give an algorithm in Section 2.2.3 which gives a placement that approximates the delay within a factor of $\frac{\alpha}{\alpha-1}$, but violates the load on each node by a factor of $\alpha + 1$. Combining this with the factor-5 loss in the reduction to the single-client case gives us Theorem 2.1.2. Finally, Section 2.3 gives efficient algorithms for placing some well-known quorum system constructions when the access strategies yielding optimal load on them are used.

2.2.1 Reduction to the Single Client Case

The following structural lemma shows that there is a *single* node v_0 in the graph G such that even if all the messages to the quorum elements were sent via the node v_0, the access delay would not increase by more than a factor of 5.

Lemma 2.2.1. *Consider any placement $f : U \to V$ of a quorum system $\mathcal{Q}$ on the nodes of G, and an access strategy p. Then there exists a vertex $v_0 \in V$ such that*

$$\mathsf{Avg}_{v \in V}[\sum_{Q \in \mathcal{Q}} p(Q) \left(d(v, v_0) + \delta_f(v_0, Q)\right)] \;\leq\; 5\mathsf{Avg}_{v \in V}[\Delta_f(v)]. \tag{2.2.4}$$

Proof. Before we begin the proof, note that the expression on the left in (2.2.4) is the average max-delay incurred if each message from v to the elements of Q first goes to v_0 (giving the $d(v, v_0)$ term) and then onto Q (giving the $\delta_f(v_0, Q)$ term).

For the proof, consider two vertices v and v' in V, and let us choose quorums Q and Q' independently at random from the distribution p. Recall that $\delta_f(v, Q) = \max_{u \in Q} d(v, f(u))$ is the maximum distance from v to any nodes in $f(Q)$. By the quorum intersection property, $Q \cap Q' \neq \emptyset$, and the triangle inequality, we get that $d(v, v') \leq \delta_f(v, Q) + \delta_f(v', Q')$. Since the quorums Q and Q' were chosen from the distribution p independently, taking expectations over the distribution we get

$$d(v, v') \leq \Delta_f(v) + \Delta_f(v').$$

Let v_0 be the node that minimizes $\Delta_f(v')$; that is, $v_0 = \text{argmin}_{v' \in V} \Delta_f(v')$. We then have:

$$d(v, v_0) \leq \Delta_f(v) + \Delta_f(v_0) \leq 2 \cdot \Delta_f(v) \tag{2.2.5}$$

(Note that, given an f, the node v_0 can be determined in time polynomial in $|V|$ just by trying all possible nodes v'.) We can now use the triangle inequality to bound the max-delay of messages sent via v_0 from the node v to the quorum Q (i.e., the "relay-via-v_0" strategy) by

$$\begin{aligned} d(v, v_0) + \delta_f(v_0, Q) &\leq d(v, v_0) + d(v_0, v) + \delta_f(v, Q) \\ &\leq 4 \cdot \Delta_f(v) + \delta_f(v, Q), \end{aligned} \tag{2.2.6}$$

where we have used (2.2.5) in the second line. Now taking expectations over p, and taking an average over $v \in V$, we get

$$\begin{aligned} &\mathsf{Avg}_{v \in V}[\sum_{Q \in \mathcal{Q}} p(Q) \left(d(v, v_0) + \delta_f(v_0, Q)\right)] \\ \leq \;& 4 \; \mathsf{Avg}_{v \in V}[\Delta_f(v)] + \mathsf{Avg}_{v \in V}[\sum_{Q \in \mathcal{Q}} p(Q) \delta_f(v, Q)], \end{aligned}$$

which simplifies to the claimed expression (2.2.4) using the definition (2.1.2) of $\Delta_f(v)$. □

Hence, even if the messages sent from each node v to quorum elements are relayed via node v_0, the resulting average delay is still less than 5 times the optimal. Moreover, the expression on the left hand side of (2.2.4), which is the average delay of this "relay-via-v_0" strategy simplifies to

$$\mathsf{Avg}_{v \in V}[d(v, v_0)] + \Delta_f(v_0). \tag{2.2.7}$$

Hence, it makes sense to try and find a placement that minimizes $\Delta_f(v_0)$, and solve the following problem:

Problem 2.2.2. (Single-Source Quorum Placement Problem (SSQPP)) *Given a quorum system $\mathcal{Q}$ over a universe U, a graph $G = (V, E)$ with a special node v_0, an access strategy p_0 with which v_0 accesses quorums in $\mathcal{Q}$ and for each node $v \in V$ an upper bound* $\mathsf{node\_cap}(v)$ *on the load it can support, find a placement $f : U \to V$ that (a) minimizes the average delay $\Delta_f(v_0)$ subject to (b)* $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$ *at each node.*

The following theorem summarizes the above discussion, and formalizes the reduction from the QPP to the Single-Source Quorum Placement Problem:

Theorem 2.2.3. *There exists a node v_0 such that a placement $f : U \to V$ that is a β-approximation for the Single-Source Quorum Placement Problem (with source v_0) is also a $5\,\beta$-approximation to the general Quorum Placement Problem.*

Proof. Consider the best placement f^* for the original Quorum Placement Problem instance, and find the node v_0 promised by Lemma 2.2.1 when given the placement f^*. Note that the placement f^* is also a solution for the Single-Source Quorum Placement Problem with special node v_0, and hence any β-approximate solution f to the Single-Source Quorum Placement Problem instance would have $\Delta_f(v_0) \leq \beta\ \Delta_{f^*}(v_0)$. Thus the delay of a "route-via-v_0" strategy with this map f would have average delay

$$\mathsf{Avg}_{v \in V}[d(v, v_0)] + \beta\ \Delta_{f^*}(v_0) \leq 5\beta \mathsf{Avg}_{v \in V}[\Delta_{f^*}(v)],$$

which follows from (2.2.4) and proves the result. □

Since we do not know the identity of the node v_0, we can run the Single-Source Quorum Placement Problem algorithm with each node in V, and pick the best placement among these.

2.2.2 NP-hardness of Problem 2.2.2

In this section, we show that the Single-Source Quorum Placement Problem is NP-hard, via a reduction from a classical NP-hard scheduling problem $1|prec|\sum w_j C_j$ (Lenstra and Kan [1978]).

Definition 2.2.4. *The input to the problem $1|prec|\sum w_j C_j$ consists of n jobs $\{J_1, J_2, \ldots, J_n\}$ with job J_j having processing time T_j and weight w_j. Furthermore, there are precedence constraints given by a partial order $\prec$ on the jobs, such that if $J_i \prec J_j$, then any feasible schedule must put job J_i before J_j. The objective is to find a feasible schedule of the jobs on a single machine so that, if the completion time of job J_j is C_j, the weighted completion time $\sum w_j C_j$ is minimized.*

In fact, a theorem from Woeginger [2001] implies that it suffices to consider only a special case of this scheduling problem:

Theorem 2.2.5. Woeginger [2001] *The following statements are equivalent:*

(a) There exists a ρ-approximation for the general problem $1|prec|\sum w_j C_j$.

(b) There exists a ρ-approximation for the special case of $1|prec|\sum w_jC_j$ where every job has either $T_j = 0$ and $w_j = 1$, or $T_j = 1$ and $w_j = 0$, and where the existence of a precedence constraint $J_i \prec J_j$ implies that $T_i = 1$ and $w_i = 0$, and that $T_j = 0$ and $w_j = 1$.

The following theorem shows a polynomial time reduction of the scheduling problem $1|prec|\sum w_jC_j$ given in Theorem 2.2.5(b) to the Single-Source Quorum Placement Problem.

Theorem 2.2.6. *The Single-Source Quorum Placement Problem 2.2.2 is NP-hard.*

Proof. Consider an instance of the scheduling problem with jobs $\{J_1, J_2, \ldots, J_n\}$; let us reorder the jobs so that all the ones with zero weight appear before those with non-zero weight. Let there be m jobs with unit weight, and hence $\{J_1, J_2, \ldots, J_{n-m}\}$ have zero weight.

For each J_j with $T_j = 1$ (and hence $w_j = 0$), let us construct an element e_j in the universe U; we add an extra element e_0 to U—hence $|U| = n - m + 1$. Let $0 < \epsilon < 1$ be a constant to be fixed later. The quorums and access strategy p are defined thus:

- For each of the m unit-weight jobs J_j (with $T_j = 0$ and $w_j = 1$), define a quorum $Q_j = \{e_0\} \cup \{e_{j'} \in U \mid J_{j'} \prec J_j\}$. Each of these quorums Q_j is accessed with probability $p(Q_j) = \frac{\epsilon}{m}$. (Call these the *type-1 quorums*.)
- For each element $u \in U$ such that $u \neq e_0$, define a quorum $Q_u = \{u, e_0\}$. Each of these quorums Q_u is accessed with probability $p(Q_u) = \frac{1-\epsilon}{n-m}$. (Call these *type-2 quorums*.)

Note that all the quorums intersect in e_0; furthermore, since there are m quorums Q_j and $(n - m)$ quorums Q_u, the access strategy p is indeed a probability distribution over the quorum system $\mathcal{Q}$. Note that the load on element e_0 is $\mathsf{load}(e_0) = 1$; for any other element $u \in U$, if u belongs to $n_u \leq m$ type-1 quorums Q_j, then its load is $\mathsf{load}(u) = \epsilon \cdot \frac{n_u}{m} + (1 - \epsilon) \cdot \frac{1}{n-m}$. Choosing ϵ such that $\epsilon < \frac{(1-\epsilon)}{n-m}$, the load for any $u \in U \setminus \{e_0\}$ is $\mathsf{load}(u) \in [\frac{1-\epsilon}{n-m}, \frac{2(1-\epsilon)}{n-m})$.

Finally, the graph $G = (V, E)$ is just a path with a node $v_j \in V$ for each element $e_j \in U$ (hence ensuring that $|V| = |U| = n - m + 1$), and edges (v_i, v_{i+1}) for $0 \leq i < n - m$ of unit length. The capacity of each node v_j with $j \neq 0$ is $\mathsf{node\_cap}(v_j) = \frac{2(1-\epsilon)}{n-m} - \epsilon$, and the capacity of v_0 is $\mathsf{node\_cap}(v_0) = 1 = \mathsf{load}(e_0)$. This completes the construction of an instance of the Single-Source Quorum Placement Problem.

Let us note some useful properties of the construction: since the capacity of any node $v_j \neq v_0$ is strictly less than $1 = \mathsf{load}(e_0)$, the element e_0 can only be placed on v_0. Moreover, since the load of any element $u \neq e_0$ lies in $[\frac{1-\epsilon}{n-m}, \frac{2(1-\epsilon)}{n-m})$, and the capacity of any $v_j \neq v_0$ is strictly less than $2 \times \frac{1-\epsilon}{n-m}$, any feasible placement must place exactly one element in U on a node of G. Furthermore, any permutation of the elements in $U - \{e_0\}$ can be placed on the vertices $V - \{v_0\}$. Thus, any placement f can be converted into a schedule in the natural way: if $f(e_j) = v_t$ for $e_j \neq e_0$ we schedule the zero-weight job J_j at time t. We also schedule unit-weight jobs $J_{j'}$ at the earliest time possible subject to the precedence constraints.

It suffices now to show that the optimal solution to the SSQPP instance gives us an optimal solution to the scheduling problem. Denote by π_f the schedule constructed from a placement f as described above. Let t_j be the time when zero-weight job J_j is scheduled and let $t_{j'}$ be the time when unit-weight job $J_{j'}$ is scheduled. The completion time of schedule π_f is then: $\mathsf{cost}(\pi_f) = \sum w_j C_j = \sum_{j':w_{j'}=1} t_{j'}$. The average delay of the placement f is:

$$\begin{aligned}
\Delta_f(v_0) &= \sum_{\text{type-1 } Q_j} p_0(Q_j) t_j + \sum_{\text{type-2 } Q_{j'}} p_0(Q_{j'}) t_{j'} \\
&= \sum_{\text{type-1 } Q_j} \frac{\epsilon}{m} t_j + \sum_{\text{type-2 } Q_{j'}} \frac{1-\epsilon}{n-m} t_{j'} \\
&= \frac{\epsilon}{m} \cdot \mathsf{cost}(\pi_f) + \frac{1-\epsilon}{n-m} \cdot \sum_{i=1}^{n-m} i
\end{aligned}$$

Now it becomes easy to see that the completion time of the schedule π_f is minimized if and only if the average delay of the placement f is also minimized. □

2.2.3 A Linear Program Rounding Solution

In light of the intractability result in Theorem 2.2.6, our goal is now to find an approximation algorithm for the Single-Source Quorum Placement Problem. To this end, we formulate Problem 2.2.2 as an integer linear program, consider its linear programming (LP) relaxation, and round this LP to get an integral solution. Unfortunately, this linear program has a large integrality gap of $O(\sqrt{n})$. However, we show that one can still get a 2-approximation algorithm from it by a *resource augmentation* argument (Kalyanasundaram and Pruhs [2000]), i.e., if we allow ourselves to violate the capacity at any node v by a small factor.

Some useful notation: since we are interested only in distances from the node v_0, let us rename nodes as $\{v_0, v_1, v_2, \ldots, v_{n-1}\}$ so that $d(v_0, v_1) \leq d(v_0, v_2) \leq \cdots \leq d(v_0, v_{n-1})$. Let us also denote $d(v_0, v_t)$ by d_t, and hence $0 = d_0 \leq d_1 \leq \ldots \leq d_{n-1}$. Let $f^* : U \to V$ be an optimal solution to Problem 2.2.2; i.e., a placement that minimizes $\Delta_{f^*}(v_0)$ subject to the constraints $\sum_{u:f^*(u)=v} \mathsf{load}(u) \leq \mathsf{node\_cap}(v)$.

To write an integer linear programming formulation for the problem, let us denote by x_{tu} the indicator for whether the element $u \in U$ is placed on the physical node v_t. Similarly, given $v_t \in V$ and $Q \in \mathcal{Q}$, the variable $x_{tQ} = 1$ indicates that all elements $u \in Q$ are placed on some subset of the nodes $\{v_0, \ldots, v_t\}$. The LP is given by:

$$\text{minimize } Z^* = \textstyle\sum_Q p_0(Q) \sum_{t=1}^{n} d_t \, x_{tQ} \tag{2.2.8}$$

$$\textstyle\sum_t x_{tu} = 1 \qquad \forall\, u \in U \tag{2.2.9}$$

$$\textstyle\sum_t x_{tQ} = 1 \qquad \forall\, Q \in \mathcal{Q} \tag{2.2.10}$$

$$\textstyle\sum_u \mathsf{load}(u) x_{tu} \leq \mathsf{node\_cap}(v_t) \qquad \forall\, v_t \in V \tag{2.2.11}$$

$$x_{tu} = 0 \qquad \forall\, u \in U,\ \forall\, v_t \in V \text{ s.t. } \mathsf{load}(u) > \mathsf{node\_cap}(v_t) \tag{2.2.12}$$

$$\textstyle\sum_{s \leq t} x_{sQ} \leq \sum_{s \leq t} x_{su} \quad \forall\, u \in Q,\ \forall\, v_t \in V,\ \forall\, Q \in \mathcal{Q} \tag{2.2.13}$$

The constraint (2.2.9) implies that each element u is assigned to one node, and (2.2.11) implies that no node v_t has too much load assigned to it. Constraints (2.2.13) and (2.2.10) ensure that if $x_{tQ} = 1$ then all the elements in Q are indeed placed on some subset of $\{v_0, \ldots, v_t\}$, and that there is indeed one such value of t. Finally, (2.2.12) ensures that no node v_t is assigned an element u with $\mathsf{load}(u)$ more than v_t's capacity $\mathsf{node\_cap}(v_t)$.

Note that if the variables are all either 0 or 1, we could get the placement $f : U \to V$ by setting $f(u) = v_t \iff x_{tu} = 1$; indeed, it is easy to see that this would be an exact formulation of the Single-Source Quorum Placement Problem. However, finding such an integral solution is NP-hard, and thus we consider the LP relaxation where the variables can take fractional values between 0 and 1: such a solution can be obtained in polynomial time. Since we have taken a relaxation of the problem, it follows that $Z^* \leq \Delta_{f^*}(v_0)$.

We now show that the integrality gap of the LP relaxation (2.2.8–2.2.13) is very large—even for distances that arise from a simple unweighted graph. This shows that that we cannot use this LP relaxation to bound the delay if we do not relax the node capacities $\mathsf{node\_cap}(v)$.

Let us first recall the definition of *Integrality Gap* (see, e.g., Vazirani [2001]). Given a linear programming relaxation for a minimization problem Π, let $LP(I)$ be the objective function value of an optimal fractional solution for the LP-relaxation. Let $OPT(I)$ denote

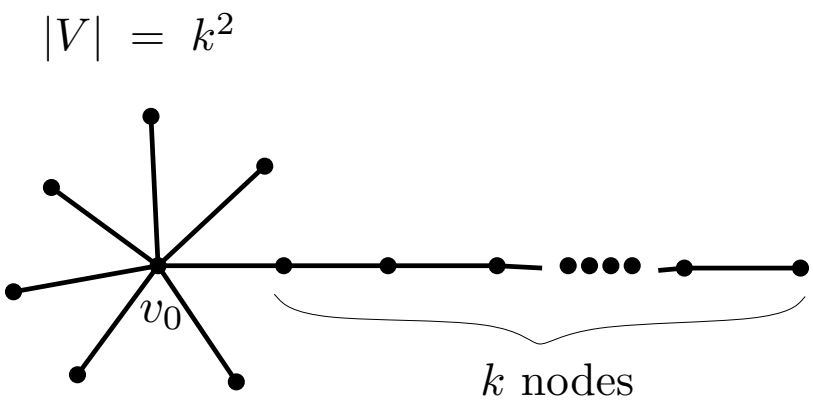


Figure 2.2.1: Graph with k^2 nodes

the cost of an optimal (integral) solution for instance I. We define the *integrality gap* (or *integrality ratio*) of the LP-relaxation to be $\sup_I \frac{OPT(I)}{LP(I)}$.

Claim 2.2.7. *The LP (2.2.8)- (2.2.13) has an integrality gap of at least n for general metric spaces, and at least $\sqrt{n}$ even for metric spaces induced by unweighted graphs.*

Proof. For the former result, consider the following instance: the quorum system $\mathcal{Q}$ has only one quorum Q containing all the n elements. The values d_t are 1 for all $0 \leq t < n-1$, and $d_{n-1} = M \gg 1$. In this case the only integral solution has $\Delta_f(v_0) = M$. However, we can set $x_{tQ} = x_{tu} = 1/n$ for all $u \in U$ and $0 \leq t < n$; the LP objective function (2.2.8) is now just $\frac{1}{n}\sum_t d_t = \frac{n-1+M}{n} \approx \frac{M}{n}$ for $M \gg n$, and hence the integrality gap is about n.

Note that some of the edges of G have non-unit length in the above example; let us now present our result for the case when the underlying graph has all edges of unit length. Again assume that we have a single quorum Q containing all the elements, and consider the graph from Figure 2.2.1 with $n = k^2$ nodes. Setting the values d_i to be the distances from v_0 to all the other nodes of the graph G, we have $d_i = 1$ for $2 \leq i \leq n-k+1$, $d_{n-k+2} = 2$, $d_{n-k+3} = 3$, …, $d_n = k$. In this case, the only integral solution has average delay k. However, if we set $x_{tQ} = x_{tj} = 1/n$ for all $1 \leq t, j \leq n$, the LP objective function has a value of

$$(n-k+1)\,\frac{1}{n} + 2\,\frac{1}{n} + \cdots + k\,\frac{1}{n} = \frac{1}{n}\left(n - k + \frac{k(k+1)}{2}\right) \approx \frac{3}{2}.$$

Thus, the integrality gap is at least $O(k) = O(\sqrt{n})$. □

Even though this LP has a large integrality gap one can still get a 2-approximation algorithm from it by a resource augmentation argument. Specifically we show how to

round the fractional solution x of LP (2.2.8–2.2.13) to obtain a map f which has an average delay at most $2Z^* \leq 2\Delta_{f^*}(v_0)$, but where the load at any node v_t is $\mathsf{load}(v_t) = \sum_{u:f(u)=v_t} \mathsf{load}(u) \leq 3\,\mathsf{node\_cap}(v_t)$. One can then generalize this result and trade off the losses in the delay and load seamlessly to get the following:

Theorem 2.2.8. *For any $\alpha > 1$, we can find a solution $f : U \to V$ to the Single-Source Quorum Placement Problem, with delay $\Delta_f(v_0) \leq \frac{\alpha}{\alpha-1}$ and load on any node $v \in V$ satisfying $\sum_{u:f(u)=v} \mathsf{load}(u) \leq (\alpha+1)\,\mathsf{node\_cap}(v)$.*

Note that the proof below is just the case $\alpha = 2$ of this theorem; the extension is not difficult, and we postpone the details until after Theorem 2.2.13.

Rounding the Fractional LP Solution

The process of rounding the fractional solution to obtain the integral solution (and hence the map f) consists of two conceptual steps. Let us give the high-level sketch before we give the details.

Filtering. In this step, we alter the LP solution to obtain a "good" (fractional) solution in which no element u is fractionally assigned to nodes that are "too far away" from v_0. Formally, after this step, if S_u is the set of nodes v_t such that $x_{tu} > 0$, then any map f satisfying $f(u) \in S_u$ will still allow us to guarantee that $\Delta_f(v_0) \leq 2Z^* \leq 2\Delta_{f^*}(v_0)$.

Rounding. We now view the good fractional solution obtained from the above step as a solution to the generalized assignment problem (GAP) and use a rounding procedure for this problem to convert the fractional solution into an integral solution such that the total load assigned to any node v_t is at most $3\,\mathsf{node\_cap}(v_t)$.

Filtering For each element u, let $\widehat{x}_{tu}$ be the largest possible value subject to the constraints that $\widehat{x}_{tu} \leq 2x_{tu}$ and $\sum_{t \leq s} \widehat{x}_{tu} \leq 1$. More precisely we set $\widehat{x}_{tu} = 2x_{tu}$ for all t's such that $\sum_{s \leq t} x_{su} < \frac{1}{2}$ and $\widehat{x}_{tu} = 1 - \sum_{s<t} \widehat{x}_{su}$ for the first t such that $\sum_{s \leq t} x_{su} > \frac{1}{2}$. We change the values $\widehat{x}_{tQ}$ similarly. (Intuitively, we are "moving mass" to the lower values of t.) Note that these new values $\widehat{x}$ satisfy (2.2.9); and they violate (2.2.11) by a factor of at most 2. Moreover, for any values of t and $u \in Q$, $\sum_{s \leq t} x_{sQ} \leq \sum_{s \leq t} x_{su}$: hence when going from x to $\widehat{x}$, either both the left and right sides of the inequality double, or the right side becomes equal to 1—in both cases, (2.2.13) holds. Note that the modified value of the objective function satisfies $\sum_Q p_0(Q) \sum_{t=1}^n d_t\, \widehat{x}_{tQ} \leq Z^*$.

We now formalize the statement that no element u is assigned to v_t which is "too far". Consider the objective function (2.2.8) of the LP: define $\overline{D}_Q = \sum_t d_t x_{tQ}$ for any quorum Q, and thus the LP value is $Z^* = \sum_Q p_0(Q)\overline{D}_Q$.

Claim 2.2.9. *All elements of quorum Q are fractionally assigned to nodes v_t with $d_t = d(v_0, v_t)$ at most $2\,\overline{D}_Q$. In other words, if $\widehat{x}_{tQ} > 0$ for some t, then $d_t \leq 2\,\overline{D}_Q$.*

Proof. This is just Markov's inequality, but here is the longer explanation. Look at the largest value t for which $\widehat{x}_{tQ} > 0$; this must be a value such that $\sum_{s<t} x_{sQ} < \frac{1}{2}$ and $\sum_{s\leq t} x_{sQ} \geq \frac{1}{2}$. If $d_t > 2\overline{D}_Q$, then Q is assigned to values larger than $d_t > 2\overline{D}_Q$ for at least a fraction of $(1 - \sum_{s<t} x_{sQ}) > \frac{1}{2}$, which violates the fact that $\overline{D}_Q$ is the average $\sum_s x_{sQ} d_s$. □

Lemma 2.2.10. *For any element $u \in U$, let $S_u = \{v_t \in V \mid \widehat{x}_{tu} > 0\}$. Then for any map f that places elements $u \in U$ on nodes in the corresponding set S_u, we have that $\Delta_f(v_0) \leq 2Z^*$.*

Proof. If $f(u) \in S_u$ then $\widehat{x}_{tu} > 0$ and so Claim 2.2.9 says that $\delta_f(v_0, Q) = \max_{u\in Q} d(v_0, f(u)) \leq 2\,\overline{D}_Q$. Therefore $\Delta_f(v_0) = \sum_Q p_0(Q)\delta_f(v_0, Q) \leq \sum_Q p_0(Q) \times 2\,\overline{D}_Q \leq 2Z^*$. □

Rounding We will now view the modified solution $\widehat{x}$ for the LP as a fractional solution to a suitable instance of the so-called *Generalized Assignment Problem* (GAP), and use techniques for that problem to round the fractional $\widehat{x}$'s to an integral solution that satisfies the assumptions of Lemma 2.2.10.

Definition 2.2.11 (GAP). *The GAP problem takes as input a set U of "jobs" and a set V of "machines", and for each $(j, i) \in U \times V$ two positive values: c_{ij} being the cost of assigning job j to machine i, and p_{ij} being the load imposed by such an assignment to machine i. The output is an assignment f of the jobs to the machines, of minimum cost $\sum_{j\in U} c_{f(j)j}$, subject to constraints on the load $\sum_{j\in f^{-1}(i)} p_{ij} \leq T_i, \forall i \in V$, given constants $T_i \in \mathbb{R}^+$ for each $i \in V$.*

Consider the following natural LP relaxation of GAP:

$$\text{minimize } Y^* = \sum_{j\in U} \sum_{i\in V} c_{ij} y_{ij} \tag{2.2.14}$$

$$\sum_{j\in U} p_{ij} y_{ij} \leq T_i \qquad \forall\, i \in V \tag{2.2.15}$$

$$\sum_{i\in V} y_{ij} = 1 \qquad \forall\, j \in U \tag{2.2.16}$$

$$y_{ij} \geq 0 \qquad \forall\, j \in U, i \in V \tag{2.2.17}$$

This relaxation was studied by Lenstra et al. [1990] and Shmoys and Tardos [1993], who proved the following result. (Here $p_i^{\max}$ is the largest load of any job assigned to machine i.)

Theorem 2.2.12. Shmoys and Tardos [1993] *Any fractional solution for the LP relaxation of GAP can be rounded into an integral solution with cost no more than Y^*, with the load on machine i being at most $T_i + p_i^{\max} \leq 2\,T_i$.*

We can use this powerful result to round our LP with the new variables $\widehat{x}$ by the following translation: the elements $u \in U$ correspond to the jobs $j = u$; the nodes $v_t \in V$ correspond to the machines $i = t$; the load p_{tu} for machine t and job u is $\mathsf{load}(u)$ if $\widehat{x}_{tu} > 0$ and $p_{tu} = \infty$ otherwise; the cost c_{tu} is the delay d_t; finally, the upper bound T_t for machine t is $2\mathsf{node\_cap}(v_t)$.

Since we ensure that no element u can be fractionally assigned to node v_t if $\mathsf{load}(u) > \mathsf{node\_cap}(v_t)$, this implies that the solution produced by applying Theorem 2.2.12 to the $\widehat{x}$'s places load at most $T_t + p_t^{\max} \leq 2\mathsf{node\_cap}(v_t) + \mathsf{node\_cap}(v_t) = 3\mathsf{node\_cap}(v_t)$. Hence the capacity is violated by at most a factor of 2, which implies the following theorem:

Theorem 2.2.13. *We can find a solution $f : U \to V$ to the Single-Source Quorum Placement Problem, where the delay $\Delta_f(v_0)$ is at most twice the LP optimum $Z^* \leq \Delta_{f^*}(v_0)$. Furthermore, the load on any node $v_t \in V$ is violated by at most a factor of three; i.e., $\sum_{u:f(u)=v_t} \mathsf{load}(u) \leq 3\,\mathsf{node\_cap}(v_t)$.*

To obtain Theorem 2.2.8, we only need to change the factor of 2 from the filtering step above to an arbitrary $\alpha > 1$. Variables $\widehat{x}_{tu}$ become the largest possible, subject to the constraints $\widehat{x}_{tu} \leq \alpha x_{tu}$ and $\sum_{t \leq s} \widehat{x}_{tu} \leq 1$. This will increase the load on each node v_t for which $\widehat{x}_{tu} > 0$ by no more than a factor of α. With the additional loss from GAP we obtain the bound of $(\alpha+1)\mathsf{node\_cap}(v)$ on the load of each node $v \in V$. The delay of any node v_t on which some element of quorum Q is placed (i.e., for which $\widehat{x}_{tQ} > 0$) becomes $d_t \leq \frac{\alpha}{\alpha-1}\overline{D}_Q$. The factor of $\frac{\alpha}{\alpha-1}$ propagates further through Lemma 2.2.10 leading to the bound on delay claimed in Theorem 2.2.8.

2.3 Optimal Layouts for Specific Constructions

In this section we address the Single-Source Quorum Placement Problem for some specific quorum systems, and give explicit placements that respect the capacities $\mathsf{node\_cap}(v)$ at the nodes while minimizing the average delay $\Delta_f(v_0)$. The specific quorum systems

considered here are the well-known **Grid** (Cheung et al. [1992], Kumar et al. [1993]) and the **Majority** (Gifford [1979], Thomas [1979]) quorum systems.

Note that the results here give us Theorem 2.1.3, since we can use the reduction of the Quorum Placement Problem to the Single-Source Quorum Placement Problem from Section 2.2.1 (with the attendant loss of a factor of 5 in the delay).

2.3.1 The Grid Construction

Consider the *Grid* quorum system (Cheung et al. [1992], Kumar et al. [1993]) on a universe U of k^2 elements. The k^2 elements are laid out on a k by k square grid M, and each quorum $Q \in \mathcal{Q}$ is formed by taking all the elements from some row and some column of M. Hence each quorum has $2k-1$ elements, and there are k^2 quorums in $\mathcal{Q}$. We assume that p_0 is the uniform access strategy, since this yields the optimal load for the Grid (Naor and Wool [1998]). Due to this uniformity, we can rephrase the objective function $\Delta_f(v_0)$ as $\sum_{i=1}^{k^2} \max_{u \in Q_i}\{d(v_0, f(u))\}$.

For simplicity, let us consider the case where the capacity $\mathsf{node\_cap}(v)$ of each node $v \in V$ is equal to the $\mathsf{load}(u)$ of any element $u \in U$ (which is the same for all elements $u \in U$ when the uniform access strategy is being used). We can easily extend our results to the general case by suppressing nodes with capacity less than $\mathsf{load}(u)$ and making multiple copies of nodes with a capacity large enough to fit multiple amounts of $\mathsf{load}(u)$ (this is equivalent to greedily packing amounts of $\mathsf{load}(u)$ into nodes with capacity $\mathsf{node\_cap}(v) \geq \mathsf{load}(u)$). The problem then becomes one of matching U to the k^2 nearest nodes to v_0; let $\tau_1 \geq \ldots \geq \tau_{k^2}$ be the distances from v_0 to these k^2 nodes to v_0 in decreasing order (i.e., the distances $d_1, \ldots, d_{k^2}$ in reverse order).

A convenient way of visualizing a placement f is to look at a $k \times k$ matrix M where each entry is one of the $\tau_1, \ldots, \tau_{k^2}$ distances; the correspondence between such matrices and placements f is given by setting $f((i,j)) = v \iff M_{ij} = d(v_0, v)$, breaking ties arbitrarily. The problem is now to place the values $\tau_1, \ldots, \tau_{k^2}$ in a $k \times k$ matrix M that minimizes the sum over all quorums of the maximum distance τ_i in each quorum. Recall that each of the k^2 quorums is the union of one row and one column of M.

The general strategy is to place the largest l^2 distances on the top-left $l \times l$ square of M. The next l distances, (i.e., τ_{l^2+1}, τ_{l^2+2}, ..., τ_{l^2+l}) are placed on positions $M_{1,l+1}$, $M_{2,l+1}$, ..., $M_{l,l+1}$, and the $l+1$ after them (i.e., τ_{l^2+l+1}, ..., τ_{l^2+2l+1}) on cells $M_{l+1,1}$, ..., $M_{l+1,l+1}$. This gets us from a $l \times l$ square to a $(l+1) \times (l+1)$ square, and having started with τ_1 in $M_{1,1}$, we can complete the placement inductively. We now prove that the

placement f obtained in this manner is optimal.

Theorem 2.3.1. *The placement f obtained by the previous algorithm is an optimal solution for the Single-Source Quorum Placement Problem on the $k \times k$ Grid when the uniform access strategy is used.*

Proof. We start with any optimal placement g and perform a number of transformations to it that do not increase its cost, at the end of which g will look as if obtained by using our strategy. This will prove that our placement strategy is optimal.

Recall that the cost of a placement for the Single-Source Quorum Placement Problem is the average delay from the client v_0 to all the quorums in the quorum system. For the case of the Grid quorum system, the delay from v_0 to a quorum Q_{ij} formed by taking the union of the elements in row i and column j, is the maximum distance $\tau_{max_{ij}}$ placed by the map g on one of the cells of row i or column j of M. Since $M_{i,j}$ uniquely determines quorum Q_{ij}, we also assign to cell $M_{i,j}$ a cost equal to the delay of quorum Q_{ij}: $\mathsf{cellcost}(M_{i,j}) = \delta_g(v_0, Q_{ij}) = \tau_{max_{ij}}$. Thus the average delay of the placement g can also be written as $\Delta_g(v_0) = \frac{1}{k^2} \sum_{i,j=1}^{k} \mathsf{cellcost}(M_{i,j})$.

Let us begin with several simple observations. First, any swapping of rows or columns of M does not change the cost of the placement g. Second, a swap of two elements $\tau_i \leq \tau_j$ placed at the intersection of two rows i and j with the same column, does not increase the cost of the placement g if τ_j is less than the maximum distance placed on row i. A similar statement can be made for elements placed at the intersection of two columns with one row.

Consider now the placement g. By swapping rows or columns we can bring element τ_1 on position $(1, 1)$ of M without changing the cost of the placement. Now assume that we have changed the position of elements $\tau_1, \ldots, \tau_{l^2}$ starting from the initial placement g to arrive at a placement consistent with our strategy and such that the cost has not increased. This implies that $\tau_1, \ldots, \tau_{l^2}$ are placed in the top left $l \times l$ square. We now show how to reposition $\tau_{l^2+1}, \ldots, \tau_{(l+1)^2}$ according to our strategy without increasing the cost.

To make the exposition easier to follow we use the following notation: row $l + 1$ and column $l + 1$ partition matrix M into four regions A, B, C, D, where A and B are the intersections of the first l rows with the first l columns and the last $k - l$ columns respectively, while D and C are intersections of the last $k - l$ rows with the first l columns and the last $k - l$ columns respectively, as in Figure 2.3.2. At the end of the previous step, each of the elements $\tau_{l^2+1}, \ldots, \tau_{(l+1)^2}$ can only be in one of the B, C, D regions. Note that all of them are less than the maximums of rows 1 through l and columns 1 through l, since we placed elements $\tau_1, \ldots, \tau_{l^2}$ in region A.

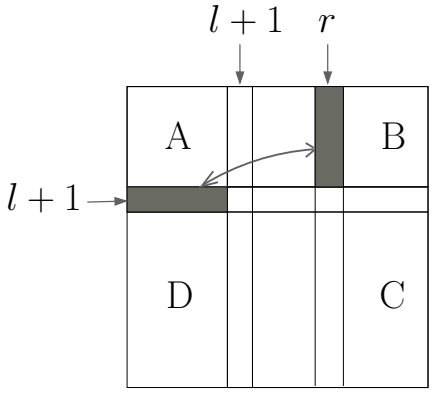


Figure 2.3.2: Partition of matrix M by row $l+1$ and column $l+1$ into regions A, B, C, D

Now look at element τ_{l^2+1}. If it is in one of the B or D regions, we can easily bring it on a cell adjacent to A by swapping rows or columns. If it is in region C, we can swap it with any of the elements from B that are on the same column without increasing the cost, since any of the first l rows has a maximum greater than τ_{l^2+1} and τ_{l^2+1} is greater than any of the elements from its column. After this move we can swap columns as before to bring τ_{l^2+1} to a cell adjacent to A. Let us assume, without loss of generality, that the position of this cell is $M_{1,l+1}$.

Consider now element τ_{l^2+2}. Assume position $M_{2,l+1}$ is occupied by $\tau_i \leq \tau_{l^2+2}$. Then swapping the two elements will not increase the cost, since placing τ_{l^2+2} on $M_{2,l+1}$ does not change the cost, while placing $\tau_i \leq \tau_{l^2+2}$ on the old position of τ_{l^2+2} might only decrease the cost. By a similar argument one can bring $\tau_{l^2+3}, \ldots, \tau_{l^2+l}$ to the right positions.

Finally, look at element τ_{l^2+l+1}. If it is in regions C or D we can bring it on a cell adjacent to A as before: either by a swap of elements that are on the same row followed by a swapping of rows (if τ_{l^2+l+1} is in region C), or just by a swapping of rows (if τ_{l^2+l+1} is in region D). If τ_{l^2+l+1} is in region B, denote by r its column, and let τ_{max} be the maximum of the elements in regions C and D. By a swap of τ_{max} with an element at the intersection of one of the first l columns with the row of τ_{max}, we can bring τ_{max} in region D without increasing the cost. By swapping rows we can then bring τ_{max} on a cell adjacent to A without changing the cost.

Now we show that by an arbitrary swapping of the elements placed at the intersection of column r with the first l rows with the elements placed at the intersection of row $l+1$ with the first l columns, the cost of the placement g does not increase (see Figure 2.3.2). Note that, by this swap, τ_{l^2+l+1} arrives on a cell of row $l+1$ adjacent to region A, as required by our placement strategy.

To compare the costs of the two placements (before and after the swap), we only need to look at the cells for which the associated cost might change after this operation. These are the cells from column r below row $l+1$ as well as the cells from row $l+1$, situated to the right of column $l+1$, with the exception of cell $M_{l+1,r}$. The sum of their costs before the swap is:

$$(k-l-1)\cdot\tau_{l^2+l+1}+(k-l-2)\cdot\tau_{max}$$

while the sum of their costs after the swap is:

$$(k-l-1)\cdot\tau_{max}+(k-l-2)\cdot\tau_{l^2+l+1}$$

which is smaller than the first one, since $\tau_{max}\leq\tau_{l^2+l+1}$.

Thus we can bring τ_{l^2+l+1} on a cell of row $l+1$ adjacent to A without increasing the cost. By direct swaps we can bring the rest of the elements up to $\tau_{(l+1)^2}$ on positions $M_{l+1,2},\ldots,M_{l+1,l+1}$ without increasing the cost. This completes the proof that g can be converted to a placement according to our strategy up to the first $(l+1)^2$ elements. By induction this shows that g can be completely converted to a placement according to our strategy without increasing the cost. Therefore our placement strategy is optimal. □

2.3.2 The Majority Construction

We now consider the following simple generalization of the well-known Majority construction. Given a universe U of size n and a parameter $t\geq\lceil\frac{n}{2}\rceil$, the quorum system consists of all the subsets of U of size t. We claim that any placement of this quorum system on the nodes of the graph has the same average delay under the uniform access strategy, which is

$$\frac{1}{\binom{n}{t}}\times\sum_{i=1}^{n-t+1}\tau_i\times\binom{n-i}{t-1}. \tag{2.3.18}$$

This is easy to see, since there are $\binom{n-1}{t-1}$ quorums containing τ_1, $\binom{n-2}{t-1}$ quorums containing τ_2 but not τ_1, $\binom{n-3}{t-1}$ quorums containing τ_3 but not the preceding two, and so on until there is a single quorum containing τ_{n-t+1} but not $\tau_1,\ldots,\tau_{n-t}$.

2.4 Total delay access cost

In this section, we present a polynomial time algorithm for the problem of minimizing the *total-delay* objective function $\mathsf{Avg}_{v\in V}[\Gamma_f(v)]$ introduced in Section 2.1, where the access

cost from a client v to a quorum Q is the sum of distances from v to all the elements of the quorum Q. This turns out to be a somewhat more tractable problem, and we can directly use techniques based on the Generalized Assignment Problem (cf. Definition 2.2.11 and Theorem 2.2.12) to give us a placement f with total-delay within a factor of two of optimum.

Theorem 2.4.1. *Consider a quorum system $\mathcal{Q}$ over a universe U with an access strategy p, and an undirected graph $G = (V, E)$ with its associated metric d and capacities $\mathsf{node\_cap}(v)$ for each $v \in V$. If $f^* : U \to V$ is a placement that minimizes $\mathsf{Avg}_{v \in V}[\Gamma_f(v)]$ subject to $\mathsf{load}_{f^*}(v) \leq \mathsf{node\_cap}(v)$, then we can find in polynomial time, a placement f with $\mathsf{load}_f(v) \leq 2\,\mathsf{node\_cap}(v)$ on $v \in V$, and where*

$$\mathsf{Avg}_{v \in V}[\Gamma_f(v)] \quad \leq \quad \mathsf{Avg}_{v \in V}[\Gamma_{f^*}(v)] \tag{2.4.19}$$

I.e., the placement minimizes the average access cost, but violates the capacity of each node by at most a factor of two.

Proof. Recall the LP relaxation 2.2.14-2.2.17 of GAP and Theorem 2.2.12. We will reduce our problem to the GAP problem, which will allow us to use the rounding technique of Theorem 2.2.12 to achieve our results.

In our problem, each element corresponds to a job in GAP (and hence we replace all j's by u's), and each graph node corresponds to a machine (thus replacing i's by v's). Also, if an element $u \in U$ is assigned to a graph node $v \in V$, its contribution to the load on v is $\mathsf{load}(u)$, and thus we set $p_{vu} = \mathsf{load}(u)$. Furthermore, the contribution to the average total-delay is $\frac{1}{n} \sum_{v' \in V} \sum_{Q : Q \ni u} p(Q) d(v', v)$, and this we set to be the "cost" c_{vu}. Of course, we set the upper bound T_v for each $v \in V$ to be the capacity $\mathsf{node\_cap}(v)$.

We can now solve the GAP LP, and use the rounding result to ensure that the cost of the resulting solution is no more than $\mathsf{Avg}_{v \in V}[\Gamma_{f^*}(v)]$, and the load placed on any node is at most $2T_v = 2\mathsf{node\_cap}(v)$, thus proving the theorem. □

2.5 Related Work

Despite being over twenty years old, research on quorum systems remains an active and rich area; see, e.g., Amir and Wool [1998], Bazzi [2000, 2001], Malkhi et al. [2000, 2001], Yu [2004] and the references therein. Previous work on quorum placement problems in graphs to minimize delays is scarcer; in particular, most previous work does not take into consideration network-oblivious measures such as load, and the natural trade-offs

arising between delay and load. Specifically, Fu [1997] introduced the following problem: given a graph $G = (V, E)$, find a quorum system $\mathcal{Q}$ over universe V to minimize $\mathsf{Avg}_{v \in V}[\min_{Q \in \mathcal{Q}} \delta(v, Q)]$, i.e., the average cost for each client to reach its "closest" quorum. That work presented optimal algorithms when G has certain characteristics, e.g., G is a tree, cycle or cluster network.

Problems of quorum design and placement on general graphs were then considered by Tsuchiya et al. [1999], who gave an efficient algorithm to find $\mathcal{Q}$ so as to minimize $\max_{v \in V} \min_{Q \in \mathcal{Q}} \delta(v, Q)$, i.e., the maximum cost any client pays to reach its closest quorum. Kobayashi et al. [2001] looked at the problem of designing quorums $\mathcal{Q}$ to minimize $\mathsf{Avg}_{v \in V}\ [\min_{Q \in \mathcal{Q}} \delta(v, Q)]$. They gave a branch-and-bound algorithm for it, which could be evaluated only on topologies with up to 20 nodes due to its exponential running time, and they also conjectured that the problem is NP-hard. Following up on this work, Lin [2001] showed that the problem is indeed NP-hard; this work, which directly motivated our research, also gives a 2-approximation for the problem.

At this point, let us mention that none of these works considered the load of the quorum system; indeed, the 2-approximation from Lin [2001] yields a quorum system with very high load—the output consists of only a *single* quorum containing a *single element* which is placed at a single node $v_0 \in V$ which minimizes $\sum_{v' \in V} d(v, v')$. Such a solution is not very desirable, since it eliminates the advantages (such as load dispersion and fault tolerance) of any distributed quorum-based algorithm. As discussed in the introduction, maintaining a low load and preserving this load dispersion capability is an essential requirement in the problems we study.

Independently of our work, Gilbert and Malewicz [2004] consider a problem they call the "partial quorum deployment problem". As in all the problems we study, their problem also takes as inputs a graph $G = (V, E)$ and a quorum system $\mathcal{Q}$ over a universe U. However, they restrict the inputs so that $|\mathcal{Q}| = |V| = |U|$, and so that each client $v \in V$ selects only a single, distinct quorum to access. In this setting, they provide a polynomial-time algorithm to compute bijections $f : U \to V$ and $q : V \to \mathcal{Q}$ that minimize $\mathsf{Avg}_{v \in V} \gamma(v, f(q(v)))$, where $f(Q) = \{f(u)\}_{u \in Q}$. They also offer a number of negative results for other variations of the Quorum Placement problem, all of which are related to $\mathsf{Avg}_{v \in V}[\Gamma_f(v)]$. Our results for the same objective function (given in Section 2.4) generalize the scenario they consider: we weaken the restrictions on the inputs, and consider more general restrictions on the load of the system.

In more distantly related work, Carmi et al. [2005] study the following problem, which we call the *geographic partition* problem: given a set $\mathcal{X}$ of n points in a closed region R of the plane, find a partition $\mathcal{Q}$ of $\mathcal{X}$ into clusters of size k so as to minimize $\max_{v \in R} \min_{Q \in \mathcal{Q}} \delta(v, Q)$. (Here, distances are in the plane.) They also address the issue

of load balancing when the geographic partition $\mathcal{Q}$ is given: assuming that the clients are uniformly distributed across the region R, the problem is to find a partition of R into subregions of equal area such that each $Q \in \mathcal{Q}$ is contained in exactly one such subregion. Carmi et al. present efficient approximation algorithms for these problems, and using techniques in Dolev et al. [2003] these can be utilized to implement intersecting quorums. However, this conversion does not preserve the delay properties of the underlying partition, and so does not solve the problem that we consider here (even in the plane).

2.6 Summary and Discussion

In this chapter we have introduced problems requiring the placement of quorums in a network so as to (approximately) minimize the average delay that clients incur to contact quorums, while (approximately) limiting the load each network node suffers to a predefined capacity. As quorums underlie numerous distributed algorithms, we believe our results are a step toward the use of such algorithms in wide-area networks, where different placements can result in varied delays for quorum access.

Numerous extensions of our results are possible. For example, a more general formulation of our Quorum Placement Problem allows clients to use different access strategies when contacting a quorum. We remark here that the proof of Lemma 2.2.1 still holds in this more general case. Furthermore, suppose that each client $v \in V$ has its own access strategy p_v. Assigning to each node an access strategy equal to the average of all the p_v's achieves the same average delay as the left-hand side of (2.2.4). Hence Theorem 2.1.2 holds for this more general version of the problem as well.

Another important observation regards the access rates made by different clients when accessing the quorum system. For the sake of simplicity we assumed these to be uniform. We emphasize here however, that our results hold even when clients use different access rates when contacting a quorum system.

Chapter 3

Quorum Placement in Networks: Minimizing Network Congestion

In this chaper we address the problem of placing an arbitrary quorum system on a network in a way that minimizes network congestion. We consider two routing models, one in which packets travel on arbitrary paths between two points (as they might, for instance, in overlay networks — we call this the *arbitrary paths* model) and one in which packets travel along a single path between two points (similar to the current Internet routing infrastructure — we call this the *fixed paths* model). In each case we show that placing quorum systems to minimize congestion is NP-hard, but that there exist approximation algorithms if one is allowed to exceed node capacities by a small factor.

We start with a description of the model and then present our main results. Sections 3.3 and 3.4 contain our treatment of the problem in the arbitrary paths model while Section 3.5 looks at the case of the fixed paths model. In Section 3.6 we study techniques of minimizing congestion in a *migration model*, where universe elements can be remapped to different network nodes over time.

3.1 Model

We model the network as an undirected graph $G = (V, E)$ of size $n = |V|$. Each physical node $v \in V$ is given a *node capacity* $\mathsf{node\_cap}(v) \in \mathbb{R}_{\geq 0}$, which is an upper bound on the amount of quorum load it wishes to handle. Furthermore, each edge $e \in E$ also has an *edge capacity* $\mathsf{edge\_cap}(e) \in \mathbb{R}_{\geq 0}$, which represents its bandwidth, i.e., the amount of traffic it can carry. For a given quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ on a universe U, an

access strategy p, and a network $G = (V, E)$, a map $f : U \to V$ placing the elements on the physical nodes is called a *quorum placement*; we will use $f(Q) \subseteq V$ to denote the set of nodes $\cup_{u \in Q}\{f(u)\}$.

We assume that the set of clients accessing the quorums in $\mathcal{Q}$ is just the node set V. To make the explanations simpler, we will assume that each of the clients $v \in V$ generates requests at some *rate* r_v with $\sum_{v \in V} r_v = 1$. It will be convenient to think of the rate r_v as the probability that client v makes a request.

We are concerned with two measures in this chapter: the *network congestion* caused by routing the requests from the clients to the nodes hosting the quorum elements, and the load generated on these nodes due to processing these requests. We use the same notion of load as the one defined in Chapter 2, which we recall here for completeness. Given a quorum system $\mathcal{Q}$ over U, an access strategy p, and a placement $f : U \to V$, the load of a network node v is defined as $\mathsf{load}_f(v) = \sum_{u \in U : f(u) = v} \mathsf{load}(u)$. Ideally, we would like placements f that satisfy $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$ for every node $v \in V$.

To define the congestion of a placement f note that a client v making an access to quorum Q needs to contact each member $f(u) \in f(Q)$ individually, and this naturally increases the traffic on the edges of some path from v to $f(u)$, for each $u \in U$. The exact congestion induced by an access depends on the routing model used:

1. In the *arbitrary routing* model, the path used for routing in the network may be chosen arbitrarily, and hence it is convenient to model traffic between any two nodes $v, v' \in V$ as a *flow* $g_{v,v'} : E \to \mathbb{R}_{\geq 0}$. (Note that each access will use a single path, but we may vary the paths used between a pair of nodes so that the average traffic on any edge is the same as in the flow.)

2. In the *fixed routing paths* model, the paths $\{P_{v,v'}\}$ are specified as part of the input, and while we can define the flow $g_{v,v'}$ as before, all the flow must travel along the path $P_{v,v'}$. This is motivated by networks like the Internet where senders and receivers cannot control or select the paths along which their traffic travels.

Thus in either model we may define the expected traffic on any edge $e \in E$ due to requests from a fixed node v to be

$$\sum_{Q \in \mathcal{Q}} p(Q) \sum_{u \in Q} g_{v,f(u)}(e)$$

Finally, since the node v is responsible for an r_v fraction of the requests, we can define the *average traffic* on the edge e to be

$$\mathsf{traffic}_f(e) = \sum_{v \in V} r_v \sum_{Q \in \mathcal{Q}} p(Q) \sum_{u \in Q} g_{v,f(u)}(e).$$

(This can be read thus: we choose the client v with probability r_v, choose a quorum Q with probability $p(Q)$, and incur a traffic of $g_{v,f(u)}(e)$ for every $u \in Q$.)

Since we are always considering averages, we will usually just refer to this as the *traffic* on the edge e. Finally, given an edge capacity $\mathsf{edge\_cap}(e)$, the *congestion* due to the placement f is

$$\mathsf{cong}_f(e) = \mathsf{traffic}_f(e)/\mathsf{edge\_cap}(e) \tag{3.1.1}$$

Ideally, this quantity should be as low as possible. Indeed, the objective function we seek to minimize is the *congestion* of the placement f, which is defined to be the congestion of the most congested edge, namely $\mathsf{cong}_f = \max_{e \in E} \mathsf{cong}_f(e)$.

Before we proceed, note that we used a flow $g_{v,v'}$ in the above discussion to model the flow of messages between v and v'. Given a placement f in the arbitrary routing model, finding a set of flows $\{g_{v,v'}\}$ that minimize the congestion (3.1.1) subject to the placement f is just a flow problem, and can be optimized in polynomial time. (Of course, the flow in the fixed-paths model is just $P_{v,v'}$.) Hence, the rest of the chapter will focus on finding the placements f: whenever we refer to a "placement f with congestion c" in the arbitrary routing model, it should be taken to read "placement f *for which there exist flows* $\{g_{v,v'}\}$ that give congestion c".

We are finally in a position to formally define the problem addressed in this chapter:

Problem 3.1.1. (Quorum Placement Problem for Congestion (QPPC)) *Given a quorum system $\mathcal{Q}$ over the universe U, an access strategy p, and an undirected network $G = (V, E)$ with capacities* $\mathsf{edge\_cap} : E \to \mathbb{R}_{\geq 0}$ *and* $\mathsf{node\_cap} : V \to \mathbb{R}_{\geq 0}$ *on edges and nodes, respectively, and client access rates $\{r_v\}$, find a placement $f : U \to V$ that (a) minimizes the congestion* cong_f *subject to (b)* $\mathsf{load}_f(v) \leq \mathsf{node\_cap}(v)$ *for all nodes $v \in V$.*

Since we can scale the capacities on the edges, we will assume (for simplicity of exposition) that the edge congestion cong_{f^*} of the optimum placement f^* is precisely 1.

Before we present our results, let us note that all our analyses apply in the *unicast* model, where an individual request is sent to *each* element of the quorum being accessed. An alternate model (which we do not consider here) would permit *multicast* messages from the source to the quorum members. Using these multicasts clearly decreases the congestion incurred: for instance, if two quorum elements are mapped to the same physical node v, these co-located elements could be reached using a single message. (Moreover, the node v could intelligently process the information reaching these co-located elements just once, thereby incurring less load.) We leave the study of these models and optimizations for future work.

3.1.1 Results

As stated, the QPPC problem turns out to be highly intractable:

Theorem 3.1.2. *Even determining whether a feasible solution for the QPPC exists (in either model) is NP hard if we do not allow any node capacities to be violated.*

We go on to show a number of approximation results for the QPPC problem if we are allowed to violate the node capacities by at most a factor of two. We use the following notation: if f^* is the optimal solution to a QPPC instance (that satisfies the load constraints), then *an* (α, β)*-approximation* is a placement f such that $\mathsf{cong}_f \leq \alpha \cdot \mathsf{cong}_{f^*}$ and $\mathsf{load}_f(v) \leq \beta \cdot \mathsf{node\_cap}(v)$ for all nodes v.

Theorem 3.1.3 (Approximations for Arbitrary Routing)**.** *For any instance of QPPC in the arbitrary routing model, we can find an* $(O(\log^2 n \log\log n), 2)$ *- approximation in polynomial time. If the graph* G *is a tree, we obtain a* $(5, 2)$*-approximation.*

Theorem 3.1.4 (Approximations for Fixed Paths)**.** *Given an instance of QPPC in the fixed routing paths model, we can find an* $(O(\frac{\eta \cdot \log n}{\log\log n}), 2)$*-approximation in polynomial time, where* η *is the size of* $\{\lfloor \log(\mathsf{load}(u)) \rfloor \mid u \in U\}$*. For example, if there exists an* N *such that* $\mathsf{load}(u) \in [1/N, 1]$ *for all* $u \in U$*, then the algorithm above yields an* $(O(\frac{\log N \log n}{\log\log n}), 2)$*-approximation.*

3.1.2 Techniques

The basis of the algorithm for the arbitrary routing model QPPC lies in a reduction of the problem to the case in which the graph is a tree *and there is only one client in the system* (i.e., there is a node v with $r_v = 1$). This reduction uses some of the properties of quorum systems, combined with the general graph decomposition result of Racke [2002]; however, it costs us a factor of $O(\log^2 n \log\log n)$ in the congestion. For the single-client tree case, we give an approximation algorithm by first writing an integer programming formulation, and then rounding its linear-programming relaxation: the rounding uses an algorithm for unsplittable flows from Dinitz et al. [1999], and is possibly of independent interest.

Our results for the fixed paths model use a different set of tools: we first develop an algorithm for instances where all the elements of U have identical ("uniform") loads. For this we use a different linear programming relaxation, and then round it using a different rounding technique that does not allow node capacities to be violated (Srinivasan [2001]). We then use this algorithm as a subroutine to solve the non-uniform case by carefully placing down sets of elements in decreasing order of their loads.

We also show hardness results for QPPC in the fixed paths model, even when the loads are all uniform. Theorem 3.5.1 states that it is NP-hard to approximate the congestion to any constant factor (even if we completely ignore the load constraints); in fact, we can obtain stronger inapproximability results under stronger complexity-theoretic assumptions.

Finally, we provide preliminary results regarding the utility of migration of universe elements between physical nodes of the network as a technique to further reduce congestion. The details of this analysis are included in Section 3.6.

3.2 Background

In this section, we introduce some concepts and results that will be used in developing algorithms for the QPPC problem in the arbitrary routing model. The "congestion preserving" trees of Racke [2002] are directly related to the problem at hand, so we discuss them in more detail in the next section. The results on unsplittable flows in Section 3.2.2 will be used in rounding a linear-programming relaxation of one of the problems we consider here.

3.2.1 Congestion Trees

Given an instance of a congestion-minimization problem on a general graph G, one may try to reduce the problem to one on a simpler graph—for instance, a tree T—where it is algorithmically easier to find a good solution. Of course, we would like that the tree T "approximates" the graph G well; the following definition formally states the notion of approximation we will use. Recall that a *multicommodity flow* on a graph $G = (V, E)$ is a set $g = \{g_i : E \to \mathbb{R}_{\geq 0}\}_i$ of flows where g_i carries d_i units from s_i to t_i ($s_i, t_i \in V$); the vector $\{d_i\}_i$ is the *value* of the flow.

Definition 3.2.1. *A tree $T = (V_T, E_T)$ with edge capacities given by $\mathsf{edge\_cap}_T : E_T \to \mathbb{R}_{\geq 0}$ is a β-approximate congestion tree for a graph $G = (V, E)$ with edge capacities $\mathsf{edge\_cap}_G : E \to \mathbb{R}_{\geq 0}$ if:*

1. *The vertices of G are the leaves of T.*
2. *For any multicommodity flow g on pairs $\{(s_i, t_i)\}_i$ that is feasible on G (i.e., $\sum_i g_i(e) \leq \mathsf{edge\_cap}_G(e)$ for each $e \in E$) there is a feasible multicommodity flow of the same value on leaves $\{(s_i, t_i)\}_i$ in T.*

3. *For any feasible multicommodity flow g_T on pairs of leaves $\{(s_i, t_i)\}_i$ in T, there exists a multicommodity flow g on $\{(s_i, t_i)\}_i$ in G such that g has the same value as g_T and $\sum_i g_i(e) \leq \beta \times \mathsf{edge\_cap}_G(e)$ for each $e \in E$.*

In a surprising result, Racke [2002] showed that one can find congestion trees for general networks with $\beta = \text{poly} \log n$. His initial result was existential, but subsequent results of Bienkowski et al. [2003], Harrelson et al. [2003] made the construction algorithmic, and also improved the value of β to give us the following theorem.

Theorem 3.2.2. *Given any undirected graph $G = (V, E)$, there exists an $O(\log^2 n \log \log n)$ - approximate congestion tree T_G; furthermore, this congestion tree can be found in time polynomial in n and the maximum capacity of any edge (assuming edge capacities are bounded to within a fixed polynomial factor of each other).*

Working in the arbitrary routing model, we will use this result to reduce an instance of the Quorum Placement Problem for Congestion on general graphs to an instance on trees, and then we will give algorithms to solve the Quorum Placement Problem for Congestion on trees.

3.2.2 Single Source Unsplittable Flow

In general, a flow from s to t could be *fractional*, i.e., the commodity travels on multiple paths from s to t. In contrast, an *unsplittable flow* is one that is constrained to travel only on a single path. The **Single-Source Unsplittable Flow Problem** (SSUFP), then, is specifically the following: given a directed graph $G = (V, E)$ with edge capacities $\mathsf{edge\_cap} : E \to \mathbb{R}_{\geq 0}$, a source node $s \in V$ and k *terminals* $t_i \in V$, with each t_i in $1 \leq i \leq k$ having a *demand* d_i, find a multicommodity flow from the source to the terminals such that the flow $g_i : E \to \mathbb{R}$ from s to t_i (of d_i units) is unsplittable (i.e., travels on a *single path*), and the total flow on any edge e is $\sum_i g_i(e) \leq \mathsf{edge\_cap}(e)$. Note that a solution to this problem is given by a set of paths $\{P_i\}_{i=1}^k$, where P_i is a path from s to t_i.

This problem was studied by Dinitz et al. [1999], who proved the following: given any feasible instance of the single-source unsplittable flow problem, there is a polynomial time algorithm to obtain a set of paths P_i (one for each terminal t_i), such that the total traffic $\sum_{i:e \in P_i} d_i$ on any edge e is at most $\mathsf{edge\_cap}(e) + \max_i\{d_i\}$. In fact, they prove a slightly stronger result, which we now state in a form most convenient to us:

Theorem 3.2.3. *Given a fractional multicommodity flow that satisfies terminal demands and the edge capacities (where the flow of d_i units from s to t_i is denoted by g_i), the*

algorithm of Dinitz et al. [1999] converts it into an unsplittable flow P_i where the total traffic over an edge e is

$$\sum_{i:e\in P_i} d_i \leq \mathsf{edge\_cap}(e) + \max\{\, d_i \mid g_i(e) > 0\}.$$

Note that the maximum on the right hand side is only over the commodities using the edge e in the input fractional flow.

In Section 3.3.2, we will use this theorem to round a fractional solution of a linear programming relaxation for the QPPC problem in the arbitrary routing model.

3.3 The Arbitrary Routing Model: The Single Client Case

In this section, we present our first results for the Quorum Placement Problem for Congestion (QPPC) in the arbitrary routing model: we consider the special case when there is *only one client* in the system generating the requests. For this case, we show that it is NP-hard to approximate the congestion within *any* factor if we enforce the node capacities $\mathsf{node\_cap}(v)$. We then show that if we are allowed to violate the node capacities by a "small" amount, we can achieve a "small" congestion as well.

3.3.1 A Hardness Result

Let us begin by proving the following simple theorem that shows that this problem is NP-hard to approximate within *any* factor. This hardness result motivates a line of inquiry we will pursue, where we allow the node capacities to be violated by a small amount, and then try to minimize the edge congestion incurred.

Theorem 3.3.1. *Finding* any *feasible solution to the Single Client case of QPPC (in either model) is NP-hard if no node capacities* $\mathsf{node\_cap}(v)$ *are violated.*

Proof. The reduction is from the PARTITION problem, an instance of which contains a set of numbers $\{a_1, a_2, \ldots, a_l\}$ with $\sum_i a_i = 2M$, and the goal is to find a subset of the a_i's that sum to exactly M.

We now construct a quorum system $\mathcal{Q}$ on $l+1$ nodes $U = \{u_0, u_1, \ldots, u_l\}$ with l quorums $Q_i = \{u_0, u_i\}$, and the access strategy $p(Q_i) = a_i/2M$. Note that $\mathsf{load}(u_0) = 1$ and $\mathsf{load}(u_i) = a_i/2M$ otherwise. Finally, let the graph $G = (V, E)$ consist of the

complete graph with 3 nodes $\{v_0, v_1, v_2\}$, with node capacities $\mathsf{node\_cap}(v_0) = 1$, and $\mathsf{node\_cap}(v_1) = \mathsf{node\_cap}(v_2) = 0.5$. (The edge capacities are not relevant in this reduction.) Finally, let all the requests originate from a single client located at v_0.

Note that any feasible placement f that respects the node capacities must place the element u_0 at the root v_0, and hence the set of elements placed at node v_1 must have $\sum_{a_i} = M$. Thus it is NP-hard to find *any feasible placement* for this instance, let alone a placement that approximates the edge congestion. □

3.3.2 The Algorithm for the Single Client Case

Our result for the special case of a single client works for the more general case of directed graphs. In fact, we also permit the presence of the following additional constraints:

- for each edge e, we can give a set of *forbidden elements* denoted by $F_e \subseteq U$ such that traffic to any element $u \in F_e$ is not allowed to traverse edge e; and
- for each node v, a set of forbidden elements $F_v \subseteq U$ that cannot be placed at the node v. (I.e., forbidden placements f are those with $f(u) = v$ for some $u \in F_v$.)

Let us denote by $\mathsf{loadmax}_v$ the maximum load of any element that can be placed on v, i.e., $\mathsf{loadmax}_v = \max_{u \notin F_v} \mathsf{load}(u)$. Similarly, let $\mathsf{loadmax}_e = \max_{u \notin F_e} \mathsf{load}(u)$. We will use these quantities to parameterize the performance of the following theorem.

Theorem 3.3.2. *Given a directed instance of the Quorum Placement Problem for Congestion in the arbitrary routing model, with a single client v_0 generating requests, let f^* be the optimal placement that respects node capacities* $\mathsf{node\_cap}$ *and achieves a congestion of* cong^* *on the edges. We can find, in polynomial time, a placement f for which:*

- *the load* $\mathsf{load}_f(v)$ *on any node v is at most* $\mathsf{node\_cap}(v) + \mathsf{loadmax}_v$*, and*
- *the traffic on any edge e is at most* $(\mathsf{cong}^* \times \mathsf{edge\_cap}(e)) + \mathsf{loadmax}_e$.

Proof. To prove this theorem, we formulate the Quorum Placement Problem for Congestion as an integer linear program (ILP), consider its linear programming (LP) relaxation, and round a (possibly fractional) solution to this LP relaxation to an integer solution to (ILP) while losing at most $O(\mathsf{loadmax}(e))$ during this rounding.

Consider the following integer linear programming formulation (ILP):

$$\lambda^* = \text{minimize } \lambda \tag{3.3.2}$$

$$\sum_i x_{iu} = 1, \qquad \forall\, u \in U \tag{3.3.3}$$

$$\sum_u \mathsf{load}(u)\; x_{iu} \leq \mathsf{node\_cap}(v_i), \qquad \forall\, v_i \in V \tag{3.3.4}$$

$$\sum_{P \in \mathcal{P}_i} g_u(P) = \; \mathsf{load}(u) x_{iu}, \qquad \forall\, u \in U, \forall\, v_i \in V \tag{3.3.5}$$

$$\sum_{P \in \mathcal{P}_i, e \in P} g_u(P) = 0, \qquad \forall\, u \in F_e, \forall\, e \in E \tag{3.3.6}$$

$$\sum_{u \in U} \sum_{v_i \in V} \sum_{P \in \mathcal{P}_i, e \in P} g_u(P) \leq \; \lambda \times \mathsf{edge\_cap}(e), \qquad \forall\, e \in E \tag{3.3.7}$$

$$x_{iu} = 0, \qquad \forall\, u \in F_{v_i} \tag{3.3.8}$$

$$x_{iu} \in \{\, 0, 1\}, \qquad \forall\, v_i \in V, \forall\, u \in U \tag{3.3.9}$$

Here x_{iu} is the indicator variable for the element u being placed on node v_i, $\mathcal{P}_i$ is the set of paths from the client v_0 to the node v_i,[1] $g_u(P)$ is the amount of traffic destined for element u that uses some path P, and λ is the overall congestion of the resulting solution. Since each of x_{iu} is either 0 or 1, the $g_u(P)$'s tell us how to send the traffic from the client v_0 to the node v_i with $x_{iu} = 1$. (Since we do not require that the $g_u(P)$'s be integral, technically the above program is a mixed-integer program.)

Note that given a solution f to the single-client QPPC problem with congestion cong_f, we may set $x_{iu} = 1 \iff f(u) = v_i$ and use the flows prescribed by the given solution to obtain $\lambda = \mathsf{cong}_f$, and hence this is indeed a formulation of the original problem.

Since we cannot solve this ILP optimally in polynomial time, we relax the integrality constraints: instead of (3.3.9), we throw in the constraint $0 \leq x_{iu} \leq 1$ and solve the resulting linear program; now we have to round the resulting fractional solution (λ, x, g) to one where $x_{iu} \in \{0, 1\}$ for all i and u. For simplicity of exposition, we scale the edge capacities by a factor of λ, so that with the new edge capacities $\lambda^* = 1$.

Preprocessing. We will use the rounding scheme used for the Single-Source Unsplittable Flow Problem to round our fractional solution, and hence we first construct an instance of SSUFP. Consider the graph $G = (V, E)$, and let us add a new "sink" vertex t to it, with directed arcs (v_i, t) from each $v_i \in V$ to this new vertex t, with each arc (v_i, t) having a capacity of $\mathsf{edge\_cap}((v_i, t)) = \mathsf{node\_cap}(v_i)$. Now we create $|U|$ new "termi-

[1]Note that $|\mathcal{P}_i|$ could be exponential in n; one can write an equivalent formulation of this ILP with a number of variables and constraints polynomial in n. However, the formulation we present here will be easier to argue about.

nals" $\{t_u \mid u \in U\}$, all of which are located at the "sink" node t. Define the client v_0 to be the "source".

Finally, note that total amount of flow ending at v_i is equal to $\sum_{u \in U} \sum_{P \in P_i} g_u(P_i) = \sum_u \mathsf{load}(u) \times x_{iu}$ using equality (3.3.5), which by (3.3.4) is at most $\mathsf{node\_cap}(v_i)$. Thus we can take all the flow that previously ended at the node v_i, and send it on the arc (v_i, t) to the sink t without violating capacities. Doing this for all vertices v_i, we get a flow that for each $u \in U$, sends $\mathsf{load}(u)$ units of flow from the source v_0 to the terminal t_u.

Using SSUFP to Round the LP Solution. Finally, we apply Theorem 3.2.3 to the flow created in the above construction: the answer it returns is a set of paths $\{P_u\}_{u \in U}$, one for each $u \in U$, such that the flow on e is

$$\sum_{u : e \in P_u} \mathsf{load}(u) \leq \mathsf{edge\_cap}(e) + \max_{u : g_u(e) > 0} \{\mathsf{load}(u)\}. \tag{3.3.10}$$

Finally, if the path P_u uses the edge (v_i, t) to reach $t_u = t$, define $f(u)$ to be v_i.

Proving the Claimed Guarantees. Let us first consider the load $\mathsf{load}_f(v_i)$, which is equal to the traffic on the arc (v_i, t). Recall that $\mathsf{edge\_cap}((v_i, t)) = \mathsf{node\_cap}(v_i)$. Also, if $g_u((v_i, t)) = \sum_{P \in \mathcal{P}_i} g_u(P)$ is non-zero, then $u \notin F_{v_i}$ by the constraint (3.3.6), and thus $\mathsf{loadmax}_{v_i} \geq \mathsf{load}_u$. Plugging these facts into (3.3.10) implies that $\mathsf{load}_f(v_i) \leq \mathsf{node\_cap}(v_i) + \mathsf{loadmax}_{v_i}$, as claimed.

Now for the traffic on an edge $e \in E$: this was originally at most $\mathsf{edge\_cap}(e)$, and now can increase by at most $\mathsf{loadmax}_e$ (due to the constraint (3.3.8)), thus proving the theorem. □

3.4 The General Case of QPPC in the Arbitrary Routing Model

To obtain the result for an arbitrary number of clients claimed in Section 2.1, we use the following strategy:

(A) Reduce the problem to trees. We first translate the QPPC problem on a general graph G to the β-approximate congestion tree T_G with $\beta = O(\log^2 n \log\log n)$, as guaranteed by Theorem 3.2.2.

It follows from the definition of a congestion tree, and the fact that the leaves of T_G correspond to nodes of the network G, that any placement $f : U \to \mathsf{leaves}(T_G)$ which is an α-approximation for the optimal congestion in T_G corresponds to a placement $f : U \to$

$V(G)$ which approximates the optimal congestion in G to within $\alpha \times \beta$. (The details of this translation are given in Section 3.4.1.)

(B) Reduce the problem to the single-source case. In Section 3.4.2, we show that there is a placement f_0 that maps all elements in U to a single node v_0 in the tree T_G and minimizes the congestion of the tree edges. However, this placement has very high load, and since our goal is to achieve low loads in addition to a low network congestion, this solution is clearly not acceptable. However, this will be a convenient structural result for the rest of the argument.

(C) Solve the single-source problem. Finally, in Section 3.4.3, we imagine the above single-node solution v_0 as a *single client generating all the requests*, and use the algorithm of Section 3.3 to find a good placement $f : U \to \mathsf{leaves}(T_G)$ for this single-client case. We show that f is also a "good" placement for the original set of clients in T_G, and achieves a congestion of $\alpha \leq 5$ times the optimum.

3.4.1 Translating the QPPC Instance to a Congestion Tree

Consider a graph $G = (V, E)$ and a β-approximate congestion tree $T_G = (V_T, E_T)$. Recall that V is equal to the set of leaves of T_G, i.e. $V = \mathsf{leaves}(T_G)$. Let $f^*_G : U \to V$ be the placement in the graph G with the least edge congestion cong^*_G. Let $f^*_{T_G} : U \to \mathsf{leaves}(T_G)$ be the placement that has the least congestion over the edges of the tree T_G, and let $\mathsf{cong}^*_{T_G}$ be the value of this congestion. By the definition of congestion trees, it follows that $\mathsf{cong}^*_{T_G} \leq \mathsf{cong}^*_G$. Since we assumed that the optimal congestion on G is exactly 1, we get the following fact.

Lemma 3.4.1. *The optimal congestion on T_G is at most* 1.

Moreover, if $f : U \to \mathsf{leaves}(T_G)$ is a placement with congestion at most $\alpha \times \mathsf{cong}^*_{T_G}$ over the edges of T_G, then f has congestion of at most $(\alpha \times \beta) \times \mathsf{cong}^*_{T_G} \leq (\alpha \times \beta) \times \mathsf{cong}^*_G$ over the edges of G. This implies the following result:

Theorem 3.4.2. *Any placement $f : U \to \mathsf{leaves}(T_G)$ with edge congestion $\alpha \times \mathsf{cong}^*_{T_G}$ over the edges of T_G has a congestion of $\alpha\beta \times \mathsf{cong}^*_G$ over the edges of G. In other words, a placement on the leaves of T_G that is an α-approximation for congestion on T_G is an $\alpha\beta$-approximation for congestion on G.*

Note that the above theorem only works for placements that map elements to the leaves of T_G, and as such cannot be used directly with the results of the next section.

3.4.2 Single Node Solutions are Good on Trees

For any node $v \in V_T$, let $f_v : U \to V_T$ be the trivial placement with $f_v(u) = v$ for all $u \in U$; i.e., all the elements of U are placed on the single node v. We will show that on a tree, an optimal placement of $(\mathcal{Q}, p)$, provided we ignore node capacity constraints, is on a single node of the tree.

Lemma 3.4.3. *Given a tree $T = (V_T, E_T)$ and a placement $f : U \to V_T$, one can find (in polynomial time) a node $v_0 \in V$ such that the placement f_{v_0} has congestion no greater than that of f.*

Proof. Let $f^{-1}(v)$ denote $\{u \mid f(u) = v\}$. For a node $v \in T$, recall that r_v was the fraction of all the requests in the system that are generated by the client v, and also that

$$\begin{aligned}\mathsf{load}_f(v) &= \sum_{u \in U : f(u) = v} \mathsf{load}(u) \\ &= \sum_{u \in U : f(u) = v} \; \sum_{Q \in \mathcal{Q} : u \in Q} p(Q) \\ &= \sum_{Q \in \mathcal{Q}} p(Q) \times |f^{-1}(v) \cap Q|\end{aligned}$$

is the expected number of messages that reach the node v (where the expectation is taken over the choice of Q under the access strategy p). It is a simple exercise to prove that there exists a node v_0 in T such that each subtree T' of $T - \{v_0\}$ has at most half the demands; i.e., $\sum_{v \in T'} r_v \leq \frac{1}{2} \leq \sum_{v \notin T'} r_v$.

Consider an edge e, and let T_L and T_R be the subtrees formed by deleting e. Let $r(T_L) = \sum_{v \in T_L} r_v$ be the total fraction of demands generated by clients in T_L. The expected number of messages seen by nodes in T_L is $\mathsf{load}_f(T_L) = \sum_{v \in T_L} \mathsf{load}_f(v)$. Let $r(T_R)$ and $\mathsf{load}_f(T_R)$ be defined similarly for the subtree T_R. Then the total congestion of the edge e under the placement f is

$$\frac{r(T_L) \times \mathsf{load}_f(T_R) + r(T_R) \times \mathsf{load}_f(T_L)}{\mathsf{edge\_cap}(e)} \tag{3.4.11}$$

Without loss of generality, let $r(T_L) \leq r(T_R)$, and hence the node v_0 must lie in T_R. Thus all the messages traversing the edge e under the placement f_{v_0} go from T_L to T_R, with e having a congestion of $r(T_L) \times [\mathsf{load}_f(T_R) + \mathsf{load}_f(T_L)]/\mathsf{edge\_cap}(e)$; the $r(T_L)\mathsf{load}_f(T_R)$ term corresponds to messages generated by nodes in T_L which are sent across e under both placements, while the $r(T_L)\mathsf{load}_f(T_L)$ term corresponds to messages generated by nodes

in T_L that are sent to nodes in T_L under placement f but are sent across e under placement f_{v_0}. Since $r(T_L) \leq r(T_R)$, this quantity is at most (3.4.11), the congestion under the placement f. Finally, we note that the node v_0 can be found in linear time simply by trying all the nodes of T, which completes the proof of the lemma. □

While this lemma tells us how to find the best quorum placement on trees, it is unsatisfying for at least two reasons. First, the node v_0 in the above theorem suffers all the load in the system under the placement f_{v_0}. Second, this node v_0 may be an internal node of T_G, and hence we cannot directly obtain a solution for the graph G by applying Lemma 3.4.3 on the congestion tree T_G, and then using Theorem 3.4.2 to translate the solution back to G. In the next section we provide a solution to these problems.

3.4.3 The Algorithm for General QPPC

Consider a congestion tree T_G, let f^* be the best placement of U on the leaves of T_G that respects the node capacities (i.e., $\mathsf{load}_{f^*}(v) \leq \mathsf{node\_cap}(v)$ for all v); let cong_{f^*} be the congestion in T_G under f^*. Let the best (single-node) placement given by Lemma 3.4.3 for the tree T_G be f_{v_0}, which places the entire quorum on v_0. Let the congestion incurred under this placement be $\mathsf{cong}_{f_{v_0}}$; Lemma 3.4.3 shows that $\mathsf{cong}_{f_{v_0}} \leq \mathsf{cong}_{f^*}$.

Let us show that if v_0 were generating all the requests (instead of the node v generating requests with probability r_v), the placement f^* would still be a fairly good placement.

Lemma 3.4.4. *The congestion incurred by the placement f^* if all the requests in the system originate at v_0 (instead of at the individual clients) is $\mathsf{cong}_{f^*,v_0} \leq 2\mathsf{cong}_{f^*}$.*

Proof. Indeed, the congestion is no worse than if we use the following routing strategy for messages: let v_0 choose Q according to the access strategy p, and a leaf v with probability r_v, and send the messages to the various nodes in $f(Q)$ by first sending them to v, which forwards them on to $f(u)$. The first part of this indirect route incurs the same congestion as the case when v were generating all the $|Q|$ messages and using the placement f_{v_0}, which is just $\mathsf{cong}_{f_{v_0}} \leq \mathsf{cong}_{f^*}$ (by Lemma 3.4.3). The second part of the route incurs a further congestion of cong_{f^*}, which proves the result. □

Recall that $\mathsf{cong}_{f^*} \leq 1$ due to Lemma 3.4.1. We now prove the main result for the QPPC problem on trees:

Theorem 3.4.5. *There is a placement f on the leaves of the tree T_G that incurs a congestion of at most $3\mathsf{cong}_{f^*} + 2 \leq 5$, and which places a load of at most $2\mathsf{node\_cap}(v)$ on each leaf v.*

Proof. Let us imagine the node v_0 of Lemma 3.4.4 to be the sole client, and use the algorithm of Section 3.3 to find a placement f on the leaves of T_G with "low" load and congestion. Each leaf node v of T_G corresponds to a node of G and hence has a node capacity already defined; for each internal node $v \in T_G$, define the $\mathsf{node\_cap}(v) = 0$, thus ensuring that no elements are mapped to internal nodes.

Recall that one could specify *forbidden sets* for nodes and edges in the algorithm of Section 3.3.2: let the forbidden set F_v for node v be the set of elements u with $\mathsf{load}(u) > \mathsf{node\_cap}(v)$. Also, the forbidden set F_e for a tree edge e is defined to be the set of all elements u such that $\mathsf{load}(u) > 2\mathsf{edge\_cap}(e)$. Note that these settings ensure that $\mathsf{loadmax}_e \leq 2\mathsf{edge\_cap}(e)$ and $\mathsf{loadmax}_v \leq \mathsf{node\_cap}(v)$.

Note that the placement f^* on the leaves of T is a possible solution to this instance of the single-client QPPC, having a congestion of at most 2 (due to Lemma 3.4.4 and Lemma 3.4.1) and load of at most $\mathsf{node\_cap}(v)$, for each $v \in V$. Hence, Theorem 3.3.2 guarantees us that (a) each node has a load of at most $\mathsf{node\_cap}(v)+\mathsf{loadmax}_v = 2\mathsf{node\_cap}(v)$, and that (b) each edge sees a traffic of at most $(\mathsf{cong}_{f^*,v_0} \times \mathsf{edge\_cap}(e)) + 2\mathsf{edge\_cap}(e)$, and hence the congestion is at most $\mathsf{cong}_{f^*,v_0} + 2 \leq 2\mathsf{cong}_{f^*} + 2$.

Now, since the requests are generated by the various nodes of the network and not by the single node v_0, one has to add in the extra congestion incurred by sending all the requests to v_0. By Lemma 3.4.3, this extra congestion is at most $\mathsf{cong}_{f_{v_0}} \leq \mathsf{cong}_{f^*}$.

Finally, putting the pieces together, the idea of conceptually "delegating" all the requests to v_0 and using the placement f that (approximately) optimizes the congestion for the "source" v_0 gives us the claimed congestion of $3\mathsf{cong}_{f^*} + 2 \leq 5$ (since $\mathsf{cong}_{f^*} \leq 1$ by Lemma 3.4.1). □

Now combined with the results of Section 3.4.1, we get the result for general graphs.

Theorem 3.4.6. *Given an instance of QPPC on general graphs, we can find a placement f that incurs on any node v a load of at most $2\mathsf{node\_cap}(v)$, and an edge congestion of at most 5β times the optimum (where β is the performance of the best known congestion tree).*

Since $\beta = O(\log^2 n \log\log n)$, this proves Theorem 3.1.3.

3.5 The Fixed Routing Paths Model

In this section we consider a variant of QPPC in which we are given routing paths $P_{v,v'}$ between each pair of vertices. A node v generating an access to element u thus incurs a

unit of flow on the edges of $P_{f(u),v}$, where u has been placed at node $f(u)$. In general, we do not require $P_{v,v'}$ and $P_{v',v}$ to be equal. As before, our goal is to find a placement f of quorum elements onto the nodes to minimize the congestion, while respecting the node capacities. First note that Theorem 3.1.2 applies to this variant; if we are not allowed to violate the node capacities then even finding a feasible solution is NP hard. As before, we retreat to the task of finding solutions that approximate the congestion well, but may violate the node capacities by a small multiplicative factor. However, even if we allow ourselves to ignore the node capacity constraints entirely (i.e., violate them by arbitrary factors), minimizing the congestion is still fairly inapproximable, as the following result states.

Theorem 3.5.1. *In the fixed routing paths model, it is NP hard to c-approximate the minimum congestion of a QPPC problem, for all $c \in \mathbb{N}$, even on instances where* $\mathsf{node\_cap}(v) = \infty$ *for all v, and* $\mathsf{load}(u) = \mathsf{load}(u')$ *for all $u, u' \in U$. Furthermore, unless* $\mathbf{NP} \subseteq \mathbf{ZPTIME}\left(n^{O\left((\log\log(n))^2\right)}\right)$*, it is NP hard to $o(\sqrt{\log\log(n)})$ - approximate the minimum congestion QPPC solution, even on instances where* $\mathsf{node\_cap}(v) = \infty$ *for all v, and* $\mathsf{load}(u) = \mathsf{load}(u')$ *for all $u, u' \in U$.*

Proof. Recall that for a vector x, $\|x\|_p := (\sum_i x_i^p)^{1/p}$, and $\|x\|_\infty = \max_i\{x_i\}$. The proof proceeds along similar lines as the proof of hardness of the Vector Scheduling problem given by Chekuri and Khanna [1999]. We reduce Independent Set to QPPC instances with $\mathsf{node\_cap}(v) = \infty$ for all v, and $\mathsf{load}(u) = \mathsf{load}(u')$ for all $u, u' \in U$. For a graph G, let $\alpha(G)$ be the size of the largest independent set in G, and let $\omega(G)$ be the size of the largest clique in G. Lemma 3.5.2 states that $\alpha(G) \geq \frac{1}{2e} n^{1/\omega(G)}$, where G has n vertices. Now consider the following multi-dimensional packing problem (MDP): given $A \in \{0,1\}^{d \times n}$ and $k \leq n$, minimize $\|Ax\|_\infty$ such that $x \in \{0,1\}^n$ and $\|x\|_1 = k$. We can reduce MDP to QPPC instances with $\mathsf{load}(u) = \mathsf{load}(u')$ for all $u, u' \in U$ in an approximation preserving fashion as follows. We construct a quorum system on k elements with uniform load. We add d vertex disjoint edges of unit capacity, one for each row of the matrix A, as well as two sources of quorum accesses, s_1 and s_2. Partition of columns of A into sets $S_1, S_2, \ldots, S_r$ using the natural equivalence relation on the column vectors, and add a vertex v_i for each S_i with $\mathsf{node\_cap}(v_i) = |S_i|$. Note that if $|S_i| = k$, we can set $\mathsf{node\_cap}(v_i) = \infty$. We also add a bottleneck edge of capacity $1/n^2$. We route the paths to ensure that placing an element at v_i is like selecting a column in S_i (add some infinite capacity edges to the graph as needed). Finally, we ensure that no elements are placed at nodes other than $\{v_1, \ldots, v_r\}$ by routing paths to these other nodes through the bottleneck edge.

Note that since we want to restrict ourselves to MDP instances that reduce to QPPC instances with uniform load and $\mathsf{node\_cap}(v) = \infty$ for all v, we require the matrix A to

satisfy the following property: if $\vec{a}$ is column vector of A, then A must have at least $k-1$ other column vectors that equal $\vec{a}$.

We now proceed by reducing Independent Set to such MDP instances. Let G be an Independent Set instance on n nodes. Fix parameters k and B. We construct a matrix A' with n columns, corresponding to each node of G. For each clique C in G of size $B+1$ or smaller, add a row C to A' such that $a'_{C,v} = 1$ if $v \in C$, and zero otherwise. Now construct a matrix A with kn columns, consisting of k copies of each column of A'. Call $x \in \{0,1\}^{kn}$ *valid* if $\|x\|_1 = k$. Note that if $\|Ax\|_\infty > 1$ for all valid x, then $\alpha(G) < k$. Furthermore, if there exists a valid x such that $\|Ax\|_\infty \leq B$, then $\alpha(G) \geq \frac{1}{2e}k^{1/B}$. To prove this, consider a graph G' that is constructed from G by replacing each node v of G with a clique C_v of size k, and adding all edges in $C_v \times C_{v'}$ to G' whenever (v,v') is an edge of G'. Clearly, $\alpha(G) = \alpha(G')$. Note that since $\|Ax\|_\infty \leq B$, the subgraph G'_x of G' induced on $\{v \mid x_v = 1\}$ has $\omega(G'_x) \leq B$, so

$$\alpha(G'_x) \geq \frac{1}{2e}|V[G'_x]|^{1/\omega(G'_x)} \geq \frac{1}{2e}|V[G'_x]|^{1/B} = \frac{1}{2e}k^{1/B} \tag{3.5.12}$$

and clearly, $\alpha(G) = \alpha(G') \geq \alpha(G'_x)$.

Given a ρ-approximation for MDP on these instances (obtained from a ρ approximation for QPPC on uniform load, infinite node capacity instances), we approximate Independent Set on G as follows. Set $k := n^{\rho/(\rho+1)}$, $B := \rho$, and construct matrix A accordingly. Let x be the output of the MDP algorithm. If $\|Ax\|_\infty > B$, output one, otherwise output $\frac{1}{2e}k^{1/B}$. The output is always at most $\alpha(G)$ by equation 3.5.12. Furthermore, in the first case $\|Ax\|_\infty > 1$ for all valid x, since we used a ρ-approximation for MDP, and thus $\alpha(G) < k$. In the latter case, $\alpha(G) \leq n$ trivially, so we have a $\max\{k, en/k^{1/B}\} = 2e \cdot (n^{1-\frac{1}{B}})$-approximation. (Note that the reduction takes $\mathrm{poly}(n^\rho)$ time.) Combining this reduction with known hardness results for Independent Set (see e.g., Engebretsen and Holmerin [2003] and the references therein), completes the proof. □

Lemma 3.5.2. *In any undirected graph G on n nodes, $2e \cdot \alpha(G) \geq n^{1/\omega(G)}$ where $\alpha(G)$ is the size of the largest independent set in G, and $\omega(G)$ is the size of the largest clique in G.*[2]

Proof. Suppose for a contradiction that $n > (2e \cdot \alpha(G))^{\omega(G)}$. Using the well known Erdös-Szekeres bound on the Ramsey number $R(s,t)$, namely $R(s,t) \leq \binom{s+t-2}{s-1}$, we conclude that

$$n > (2e \cdot \alpha(G))^{\omega(G)} \geq \binom{\alpha(G)+\omega(G)}{\omega(G)} \geq R(\alpha(G)+1, \omega(G)+1)$$

[2]We note that stronger versions of this lemma exist, and a similar lemma is stated without proof in Chekuri and Khanna [1999], however this version is sufficient for our purposes.

Thus, by the definition of $R(\cdot,\cdot)$, G has an independent set of size $\alpha(G)+1$ or a clique of size $\omega(G)+1$, which yields the desired contradiction. □

We now develop an approximation algorithm for QPPC in the fixed paths model, starting with instances with uniform element loads.

3.5.1 Uniform Element Loads

Theorem 3.5.3. *There is a polynomial time randomized algorithm that, given an instance of the QPPC problem in the fixed routing paths model in which* $\mathsf{load}(u) = \mathsf{load}(u')$ *for all* $u, u' \in U$*, yields a* $(O(\log n/\log\log n), 1)$*-approximation.*

We reformulate the QPPC problem in the fixed paths model with uniform elements loads as follows. Assume WLOG that for each $u \in U$, $\mathsf{load}(u) = l$. Consider placing a logical element u at a node v. Since the loads are uniform, placing any logical element at v results in the same increase in congestion to the edges of the network. We represent this as a vector $c_v \in \mathbb{R}^{|E|}$, where the coordinates are indexed by edges. Thus coordinate e of c_v is the expected congestion incurred by placing an element at v. For each v, suppose we can place at most $h(v) := \left\lfloor \frac{\mathsf{node\_cap}(v)}{l} \right\rfloor$ logical elements at v while respecting the node capacities. Consider a matrix A that has exactly $b := \sum_v h(v)$ columns, consisting of $h(v)$ copies of c_v for each v. We say these $h(v)$ columns are *associated* with v. Our variant of the QPPC problem thus becomes

$$\textbf{minimize} \quad \|Ax\|_\infty \quad \textbf{s.t.} \quad x \in \{0,1\}^b \text{ and } \|x\|_1 = |U|$$

We say that x *selects* columns i for which $x_i = 1$, and for each column associated with v that x selects, we place a logical element at v. We call the resulting assignment f_x. It is easy to encode this formulation as an ILP and take the LP relaxation.

$$\begin{array}{rcl} \lambda^* & = & \text{minimize } \lambda \\ \lambda & \geq & \sum_j a_{ij} x_j \ \ \forall i \\ \sum_j x_j & = & |U| \\ x_j & \in & [0,1] \ \ \forall j \end{array}$$

To solve this LP we can start by guessing the optimal congestion [3] cong^*, and remove

[3] If guessing cong^* requires too much nondeterminism, it is sufficient to guess $t = \left\lceil \log_{(1+\epsilon)}(\mathsf{cong}^*) \right\rceil$, for any $\epsilon > 0$, and use $(1+\epsilon)^t$ as an estimate for cong^*. This increases the bound on congestion by a factor of $1+\epsilon$.

all columns containing any entry $a_{ij} > \mathsf{cong}^*$ from the matrix A. We then solve the resulting LP, and apply the rounding scheme of Srinivasan [2001] to the resulting optimal fractional solution x to get an integral vector y.

Using this rounding procedure, Srinivasan guarantees that $\|y\|_1 = |U|$, and for all vectors a such that $a_j \in [0,1]$ for all j, and for all $\delta \geq 0$ and $\mu \geq \mathbf{E}\left[\sum_j a_j y_j\right]$

$$\mathbf{Pr}\left[\sum_j a_j y_j \geq \mu(1+\delta)\right] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \tag{3.5.13}$$

As before, we can scale the values a_{ij} by $1/\mathsf{cong}^*$, so that the optimal congestion becomes one, and each $a_{ij} \leq 1$. We can apply then equation 3.5.13 to bound the congestion on a fixed edge i. Note that $\mathbf{E}\left[\sum_j a_{ij} y_j\right] = \sum_j a_{ij} x_j \leq 1$, since the optimal congestion is one, so we set $\mu = 1$. For any constant c, we can apply equation 3.5.13 with some $\delta = \Theta(\log n / \log\log n)$ to prove that the congestion on edge i exceeds the optimal congestion by more than an additive factor of δ with probability at most $1/n^c$. Taking a union bound over the edges, we infer that the congestion is $O(\log n / \log\log n)$ with high probability. Thus the placement f_y is a $(O(\log n / \log\log n), 1)$-approximation.

The algorithm is summarized as follows:

Algorithm for uniform load instances:
Generate matrix A and guess cong^*.
Remove columns j of A with $\max_i\{a_{ij}\} > \mathsf{cong}^*$.
Optimally solve the resulting LP to get solution x.
Round x to y using the rounding in Srinivasan [2001].
Output f_y.

3.5.2 The General Case

Here, let $\mathcal{A}$ be any algorithm for uniform load instances. If $\mathcal{A}$ is the algorithm given above, we suppose it is given its guess for cong^* as part of its input.

Algorithm for general instances:
Guess $\kappa = \mathsf{cong}^*$.
For each $u \in U$, round $\mathsf{load}(u)$ down to the nearest power of two. Call the result $\mathsf{load}'(u)$.
Let $L := \{\mathsf{load}'(u) \mid u \in U\}$.
For each $l \in L$, in decreasing order of size
    Run $\mathcal{A}$ on $U_l := \{u \in U \mid \mathsf{load}'(u) = l\}$,
    using κ as the input guess if needed.
    Place U_l as $\mathcal{A}$ suggests, and decrease $\mathsf{node\_cap}(\cdot)$ accordingly. That is, if t elements of U_l are placed on v, decrease $\mathsf{node\_cap}(v)$ by tl.

Lemma 3.5.4. *If $\mathcal{A}$ is a (α, β)-approximation for QPPC instances with uniform load in the fixed routing paths model, then the above algorithm is a $(\alpha|L|, 2\beta)$-approximation for general QPPC instances in the fixed routing paths model.*

Proof. Suppose f is the placement output by the algorithm. We first prove $\mathsf{load}_f(v) \leq 2\beta\mathsf{node\_cap}(v)$ for each v. Note that it suffices to show that $\mathsf{load}'_f(v) \leq \beta\mathsf{node\_cap}(v)$, since $\mathsf{load}(u) \leq 2\mathsf{load}'(u)$ for all v. From now on all references to load refer to load'.

Fix any v. Suppose $\mathcal{A}$ is run on elements u with $\mathsf{load}'(u) = l$ and places t of them on v. There are two cases: either at this stage, $\mathsf{node\_cap}(v) \geq tl$, in which case we can charge the load these elements cause to the corresponding decrease in $\mathsf{node\_cap}(v)$, or else $\mathsf{node\_cap}(v) < tl$. In the latter case, we know that $\mathsf{node\_cap}(v) \geq tl/\beta$ since $\mathcal{A}$ is an (α, β) - approximation, so we can charge tl/β to $\mathsf{node\_cap}(v)$. Furthermore, since $\mathsf{node\_cap}(v)$ is reduced to zero, v is not assigned any additional elements later on. Thus we can charge $1/\beta$ of the load to $\mathsf{node\_cap}(v)$, and conclude that $\mathsf{load}'_f(v) \leq \beta\mathsf{node\_cap}(v)$.

We now bound the congestion caused by each execution of $\mathcal{A}$ by $\alpha \cdot \mathsf{cong}^*$. To do this, it suffices to prove that the optimal congestion is at most cong^* in each instance on which $\mathcal{A}$ is run. Fix an instance, say on elements with $\mathsf{load}'(u) = l$, denoted U_l. All elements with larger loads have already been placed, thus reducing the node capacities at some nodes. For a placement f, node v, and $W \subseteq U$, let

$$\mathsf{cap}(f, v, W) \;=\; \mathsf{node\_cap}(v) \;-\; \sum_{u \in W : f(u) = v} \mathsf{load}'(u)$$

denote the remaining load at v after placing down W using f. Here, $\mathsf{node\_cap}(v)$ are the original input node capacities. Let $U' := \{u \mid \mathsf{load}'(u) \geq 2l\}$, and let f be the partial placement of U' created by the algorithm so far. Fix any optimal solution f^*. We can

place down elements of U_l at nodes v such that $\mathsf{cap}(f, v, U') - \mathsf{cap}(f^*, v, U' \cup U_l) > 0$. Specifically, we place down $\lfloor(\mathsf{cap}(f, v, U') - \mathsf{cap}(f^*, v, U' \cup U_l))/l\rfloor$ such elements at v. It remains to show that we can place all of U_l down this way. To see this, first note that, by a simple volumetric argument,

$$\sum_v (\mathsf{cap}(f, v, U') - \mathsf{cap}(f^*, v, U' \cup U_l)) = l \cdot |U_l|$$

Next, observe that $\mathsf{cap}(f, v, U') - \mathsf{cap}(f^*, v, U' \cup U_l)$ is always a multiple of l, since all elements of U' and U_l have loads that are multiples of l. (Note how we have used the fact that the loads $load'(u)$ are multiples of two.) Combining these two facts, we see that we can pack U_l in node capacity occupied by elements of $U' \cup U_l$ under placement f^*, while respecting node capacity constraints (with respect to load'), no matter how f placed down U'. Having done this, it is clear that the resulting congestion due to placing U_l is no more than cong^*.

Since the congestion due to $\mathcal{A}$ on each instance is at most $\alpha \cdot \mathsf{cong}^*$, we conclude that all executions of $\mathcal{A}$ together contribute congestion at most $|L| \cdot \alpha \cdot \mathsf{cong}^*$. □

Note that $|L| = |\{\lfloor \log_2(\mathsf{load}(u))\rfloor \mid u \in U\}| = \eta$, so using the algorithm given above for $\mathcal{A}$, with $\alpha = O(\log n / \log\log n)$ and $\beta = 1$, we complete the proof of Theorem 3.1.4.

3.6 The Migration Model

In this section we study the congestion of a quorum system placement in a variant of the arbitrary routing model. We assume that the logical elements of U can *migrate* from one physical node to another. For simplicity we ascribe zero cost to the migration of logical elements, leaving as future work the study of the problem in a model with non-zero costs for migration.

Our objective function is the congestion of the most congested edge $e \in E$ amortized over Δ time units, where each element $u \in U$ is stationary during each time unit (and can migrate in between). A solution to this problem is a *placement with migration*, i.e., a function $h : U \times \{1, \ldots, \Delta\} \to V$, where $h(u, t)$ specifies the node v that hosts u during time unit t. No bounds are placed on the capacity of any physical node, in other words, load is not an issue here. As with migration cost, we leave the problem of addressing load in a migration model as future work.

We now give an example which shows that, in arbitrary graphs, migration can indeed help reduce congestion. Consider the complete graph K_n on n vertices, with each edge

having unit capacity, and assume that the universe of logical elements consists of a single node, $U = \{u\}$. A static strategy would specify a placement $f : U \to V$ of u on $v = f(u)$, one of the nodes of K_n. Assuming that each client sends a request each time unit, the amortized congestion of the placement is 1.

Consider now what happens when we allow migration. Suppose that after each client request we move the logical element from one physical node to the next in a circular manner such that all nodes are used. In this case all edges have congestion $1/\frac{n}{2}$ which is less than the one obtained for a fixed placement. In fact, a simple averaging argument shows that $O(n)$ is the largest gap that can be obtained between the congestions of the two models (with and without migration).

This example indicates that studying the model in which migration is allowed can have possible benefits in terms of congestion. Unfortunately, this is not true for all graphs, in particular, it is not true for trees, as we will now prove.

Lemma 3.6.1. *For a tree T there exists a node v_0 such that no placement with migration h of a quorum system over the universe with a single element $U = \{h\}$ can have a congestion better than that of h_{v_0}, where $h_{v_0}(u, t) = v_0$ for each $t \in \{1, \ldots, \Delta\}$.*

Proof. The proof is similar to that of Lemma 3.4.3. Let $h : U \times \{1, \ldots, \Delta\} \to V$ be an arbitrary placement with migration, and let $h_t = h(\cdot, t) : U \to V$ be the placement specified by h at time t. For an edge $e \in E$, let $r(T_L)$ and $r(T_R)$ be the request rates of clients coming from the two subtrees T_L and T_R (obtained by removing e from T). Let also $\mathsf{load}_{h_t}(T_L) = \sum_{v \in T_L} \mathsf{load}_{h_t}(v)$ and $\mathsf{load}_{h_t}(T_R)$ defined similarly, be the expected number of messages seen by nodes in T_L and T_R respectively, at time t. Then the congestion of the edge e over the time period Δ is

$$\sum_{t=1}^{\Delta} \frac{r(T_L) \times \mathsf{load}_{h_t}(T_R) + r(T_R) \times \mathsf{load}_{h_t}(T_L)}{\mathsf{edge\_cap}(e)} \tag{3.6.14}$$

Let v_0 be the node found in Lemma 3.4.3 and assume that $r(T_L) \leq r(T_R)$. Node v_0 has to lie in T_R and thus the congestion of e for the placement h_{v_0} with migration is $\frac{\Delta \times r(T_L) \times \left(\mathsf{load}_{h_{v_0}}(T_R) + \mathsf{load}_{h_{v_0}}(T_L)\right)}{\mathsf{edge\_cap}(e)}$. Since $r(T_L) \leq r(T_R)$ and the total load of the system does not change, this is at most the quantity given by 3.6.14, which completes our proof. □

3.6.1 A solution for arbitrary graphs.

To obtain a solution for arbitrary graphs we will use Räcke's results on congestion trees. Consider an arbitrary graph G and construct its associated congestion tree T_G. Then find the node v_0 from Lemma 3.4.3 that minimizes congestion, assuming the request rates of clients are known. If the node v_0 is a leaf we are done, we can simply use the placement f_{v_0} in the original graph G with only a $\mathrm{polylog}\, n$ loss in congestion. If v_0 is an internal node in T_G we need to specify a way in which v_0 gets mapped to one of the nodes of G in the cluster corresponding to v_0 in G. In Räcke's work this was done by choosing the leaf onto which v_0 is mapped, independently at random from a special distribution depending on the cluster corresponding to v_0. More precisely, each leaf was chosen with a probability proportional to its *weight* in that cluster (which was equal to the sum of the capacities of the edges incident to that node that were leaving the cluster). This is done independently at random for each message that is routed through the node v_0.

To obtain the same approximation ratio for congestion, we can do something similar here (this is based on ideas from Westermann [2001]). After a fixed amount of time, the node in the cluster of v_0 onto which v_0 is mapped, makes a decision as to whether it should keep all the logical elements of U mapped onto itself or it should migrate them to another node of the cluster corresponding to v_0. The next node in the migration chain is picked independently at random from the special distribution mentioned before from the nodes of the cluster. This ensures that over a longer period of time, we will match the conditions that enable Räcke's construction to provide the $\mathrm{polylog}$ approximation factor for congestion. By an argument similar to the one from Section 3.4.1, we can see that our solution will also suffer only a $\mathrm{polylog}$ loss in congestion compared to the optimal one in the migration model, regardless of whether that solution uses migration or not.

Here is an example illustrating how our algorithm works for a particular graph. Consider the congestion tree T_{K_n} for the complete graph K_n and assume that all edges of K_n have unit capacity. Assume further, that clients issue requests uniformly from all the nodes of K_n. The tree T_{K_n} will consist of a root and n leaves, each leaf being connected to the root by an edge of capacity $n-1$. The algorithm will find the root as the node minimizing congestion and will place all the elements of U on it. The root is mapped to one of the leaves with probability $\frac{1}{n}$ and then migrated after some fixed amount of time to a new leaf chosen independently at random (and uniformly in this case) from all the leaves of T_{K_n}. This, in fact, corresponds to the optimal solution for the complete graph K_n.

3.7 Related work

To the best of our knowledge, previous work on quorum placement in networks has only considered minimizing various notions of delay (as discussed in 2.5). Minimizing congestion for quorum placements is a less studied problem. On the other hand minimizing congestion for both specific and general networks is a problem that has received considerable attention in the past; given the impossibility of summarizing this work, we mention just some of the most important results here. Early work in this area included the seminal results of Valiant [1982] and Valiant and Brebner [1981] who gave randomized routing algorithms in hypercubes and meshes to get small congestion. Leighton et al. [1989] then gave deterministic algorithms for meshes. Linear programming relaxations and randomized rounding was first used by Raghavan and Thompson [1985] to find unsplittable paths with low congestion. Single-source versions of unsplittable flow were studied by Dinitz et al. [1999], who gave constant-factor approximation algorithms for various versions of the problem.

In a model similar to ours, Maggs et al. [1997] consider a data management problem for special networks (trees, meshes, and clustered networks). In their work, clients issue read and write requests for objects, where a read request is serviced by any node holding a copy of the object, but a write request must update all copies of the object. Similar to our case their goal is to place the objects optimally on the nodes of a network to minimize congestion. However, while their paper considered the questions behind replicating objects and the static and dynamic issues therein (i.e., how many copies of an object to maintain at any time? where to place them?), here we take a fixed quorum system and client request rates as input and try to find congestion-optimal placements that respect node capacities.

The results of Maggs et al. [1997] are extended by Westermann [2001] to a model in which objects are allowed to *migrate* between nodes of the network: while migrating an object increases congestion, moving the object closer to a source may eventually decrease traffic in the network. He gives a 3-competitive algorithm for congestion for trees, and extends these results to other classes of networks.

Racke [2002] further generalizes these results by giving a general method to solve a congestion problem in arbitrary graphs. His method is based on the construction of a *congestion-tree* T_G that "simulates" the original graph with a $\mathrm{polylog}\ |V|$ factor loss in congestion; more details on this general method are given in Section 3.2.1.

3.8 Conclusions

In this chapter we studied the problem of placing the elements of a universe U underlying a quorum system $\mathcal{Q}$ on a network G in a way that minimizes congestion due to quorum accesses, while respecting the computing capacity of each network node. We considered this problem in two models, differing on the basis of whether communication routes are fixed or can be chosen. We showed that in either case, this problem cannot be approximated to within *any* factor (unless P=NP). However, by allowing doubling of the capacity of each node, we present efficient approximation algorithms for this problem in both models. We have also shed some light on the extent to which element migration can reduce congestion in this context.

Chapter 4

Minimizing Response Time for Quorum-System Protocols over Wide-Area Networks

In this chapter we consider the use of quorum-based protocols in a wide-area network. Our interest in the wide-area setting arises from two previous lines of research: First, quorums have been employed as an ingredient of *edge computing* systems (e.g., Gao et al. [2005]) that support the deployment of dynamic services across a potentially global collection of proxies. That is, these techniques strive to adapt the efficiencies of content distribution networks (CDNs) like Akamai to more dynamic services, and in doing so they employ accesses to quorums of proxies in order to coordinate activities. Second, there has recently been significant theoretical progress on algorithms for deploying quorums in physical topologies, so that certain network measures are optimized or approximately optimized (e.g., Fu [1997], Golovin et al. [2006], Gupta et al. [2005], Kobayashi et al. [2001], Lin [2001], Tsuchiya et al. [1999]). These results have paved the way for empirical studies using them, which is what we seek to initiate here.

Our main goal in this chapter is to perform an evaluation of techniques for placing quorums in wide-area networks, and for adapting client[1] strategies in choosing which quorums to access. In doing so, we shed light on a number of issues relevant to deploying a service "on the edge" of the Internet in order to minimize service response times as measured by clients, such as (i) the number and location of proxies that should be involved in the service implementation, and (ii) the manner in which quorums should be accessed. A central

[1]While we refer to entities that access quorums as "clients", they need not be user end systems. Rather, in an edge computing system, the clients could be other proxies, for example.

tension that we explore is that between accessing "close" quorums to minimize network delays and dispersing service demand across (possibly more distant) quorums to minimize service processing delays. Similarly, as we will show, quorums over a small universe of servers is better to minimize network delays, but worse for dispersing service demand. Finding the right balances on these spectra is key to minimizing overall response times of a edge-deployed service.

We conduct our analyses through both experiments with a real protocol implementation (Abd-El-Malek et al. [2005]) in an emulated wide-area network environment (Vahdat et al. [2002]) and simulation of a generic quorum system protocol over models of several actual wide-area network topologies. The topologies are created from PlanetLab (Bavier et al. [2004]) measurements and from measurements between web servers (`http://ww.icir.org/tbit/daxlist.txt`). Their sizes range from 50 to 161 wide-area locations, making this, to our knowledge, the widest range of topologies considered to date for quorum placement. That said, as the initial study at this scale, ours is limited in considering only "normal" conditions, i.e., that there are no failures of network nodes or links, and that delays between pairs of nodes are stable over long enough periods of time and known beforehand. We hope to relax these assumptions in future studies.

The rest of the chapter is organized as follows: in Section 4.1 we perform experiments with a quorum protocol implementation on an emulated network topology with the goal of determining how network delay and server load affect client response time in a wide-area environment. In Section 4.2 we describe several algorithms designed to minimize response time by balancing load and network delay. These algorithms are based on the assumption that client demand is known in advance. In the next chapter we also consider the case of dynamic workloads. In Sections 4.4 and 4.5 we evaluate these algorithms through simulations over several wide-area topologies. In Section 4.7 we discuss related work. We present conclusions about our findings in Section 4.8.

4.1 A motivating example

To motivate our study (and perhaps as one of its contributions), in this section we describe an evaluation of the Q/U protocol (Abd-El-Malek et al. [2005]). Q/U is a Byzantine fault-tolerant service protocol in which clients perform operations by accessing a quorum of servers. The goal of our evaluation is to shed light on the factors that influence Q/U client response time when executed over the wide area, leading to our efforts in subsequent sections to minimize the impacts of those factors.

We perform our evaluation on Modelnet (Vahdat et al. [2002]), an emulated wide area

environment, using a network topology developed from PlanetLab measurements. We chose to run Q/U on Modelnet as opposed to Planetlab, since Modelnet is a better environment for producing repeatable results. We deployed 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 derived our topology from network delays measured between 50 PlanetLab sites around the world in the period July–November 2006 (see `http://ping.ececs.uc.edu/ping/`).

We varied two parameters in our experiments: the first was the number n of Q/U servers, where $n \geq 5t + 1$ is required to tolerate t Byzantine failures. We ran Q/U with $t \in \{1, \ldots, 5\}$, $n = 5t + 1$ and a quorum size of $4t + 1$. The second parameter we varied was the number of clients issuing requests to the Q/U servers. We chose the location of clients and servers in the topology as follows: we placed each server at a distinct node, using the placement algorithm from Section 2.3.2. We then computed a set of 10 client locations for which the average network delay to the server placement approximates the average network delay from all the nodes of the graph to the server placement well. On each of these client locations we ran c clients, with $c \in \{1, \ldots, 10\}$.

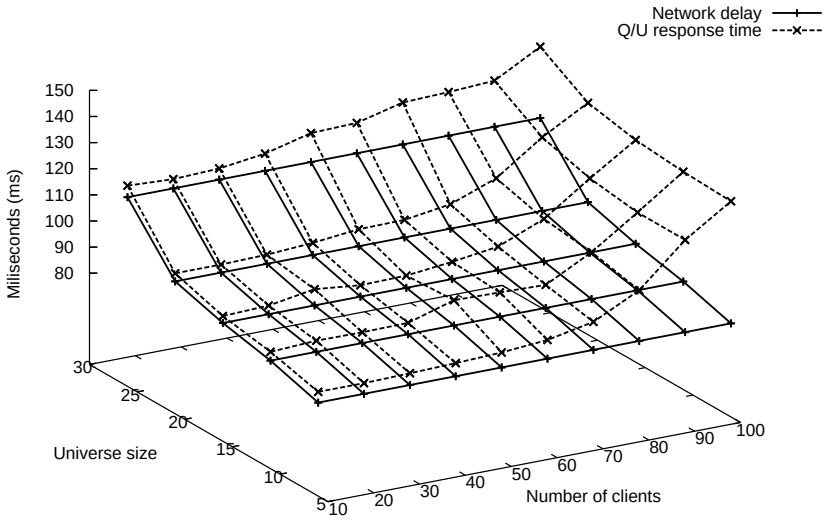


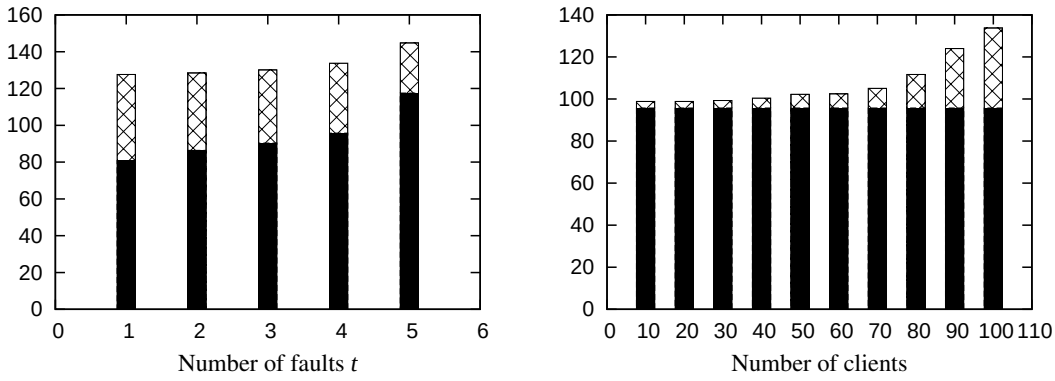
Figure 4.1.1: Average response time, network delay for Q/U on Planetlab topology

Clients issued only requests that completed in a single round trip to a quorum. While not all Q/U operations are guaranteed to complete in a single round trip, they should in the common case for most services (Abd-El-Malek et al. [2005]). For each request, clients

chose the quorum to access uniformly at random, thereby balancing client demand across servers. The application processing delay per client request at each server was 1 ms.

We compared two measures in our experiments: the average response time over all the clients and the average network delay over all the clients (both in milliseconds). Average response time was computed by running each experiment 5 times and then taking the mean. The variation observed was under 1 ms for up to 50 clients, and then increased with the client demand above that threshold.

Figure 4.1.1 shows how the two measures changed when we varied the universe size n and the number of clients issuing requests. As expected, increasing the client demand led to higher processing delay and hence higher average response time. Increasing the universe size had a similar effect on response time, but for a different reason: the average network delay increased since quorums tended to be spread apart more. This can be more easily seen in Figure 4.1.2a, where we keep the client demand constant and increase t and hence the universe size. However, increasing the universe size better distributed processing costs across more servers, and so decreased processing delay slightly.



(a) 100 clients (b) $t = 4$, $n = 21$

Figure 4.1.2: Avg network delay (black bars) & response time (total bars) for Q/U (ms)

In Figure 4.1.2b we can see how putting more demand on the system increased the average response time. For up to at most 50 clients, the major component of the average client response time was network delay. Increasing client demand beyond that point made processing delay play a more important role in average client response time (although network delay still represented a significant fraction of the overall response time). If request processing involved significant server resources, this effect would be even more pronounced.

To summarize, the Q/U experiments show that on a wide-area network, average response time depends on the average network delay and on the processing delays on servers. Increasing universe size tends to increase network delay but decrease per-server processing delay when demand is high. Thus to improve the overall performance of a protocol that uses quorum systems, we need algorithms that optimize either just the network delay (for systems where client demand is expected to be low) or a combination of network delay and processing delay (if client demand is expected to be high). The rest of this chapter is devoted to balancing these tradeoffs by modifying how servers are placed in the network, and how clients access them.

4.2 Algorithms

To experiment with quorum placement in wide area topologies, we have implemented some of the algorithms presented in Chapter 2 and developed others of potentially independent interest. For completeness we describe here all the algorithms and insist only on details relevant to our evaluation. We start with a description of the model (which is an extension of the model in Chapter 2).

4.2.1 Model

We model the network as an undirected graph $G = (V, E)$, with each node having an associated capacity $\mathsf{node\_cap}(v) \in \mathbb{R}^+$ and each edge having a positive "length" $\mathsf{length}(e)$. The length function induces a distance function $d : V \times V \rightarrow \mathbb{R}^+$ obtained by setting $d(v, v')$ to be the length of the shortest path between v and v'. We assume that the set of clients that access the quorum system is V.

Given a quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ over a universe U and a placement $f : U \rightarrow V$, we define $f(Q) = \{f(u) : u \in Q\}$, for an arbitrary $Q \in \mathcal{Q}$. We call $f(U)$ the *support set* of the placement f (i.e., the nodes of the graph that actually host a universe element).

We assume that clients can choose quorums based on different access strategies. We denote by p_v the access strategy of client $v \in V$ and by $\mathsf{load}_v(u) = \sum_{Q \in \mathcal{Q}: u \in Q} p_v(Q)$, the load induced by this access strategy on element $u \in U$. When a placement f is specified, we can define the load induced by a client $v \in V$ on a network node $w \in V$ as $\mathsf{load}_{v,f}(w) = \sum_{u \in U: f(u) = w} \mathsf{load}_v(u)$ (i.e., equal to the sum of load of universe elements assigned to w by f). Finally, we define the total load induced by f on node w as

$\mathsf{load}_f(w) = \mathsf{avg}_{v \in V} \mathsf{load}_{v,f}(w)$.

The objective function we seek to minimize in this chapter is the *average client response time*. To define it we start with a definition of the response time for a single client request:

$$\rho_f(v, Q) = \max_{w \in f(Q)} (d(v, w) + \alpha * \mathsf{load}_f(w)) \tag{4.2.1}$$

This can be read as follows: the service time of a client request is the time the request spends on the network (which we assume here is just the network delay) plus the amount of time spent at the server (proportional to the load of the server) scaled by a factor α. Manipulating α allows us to adjust for absolute demand from clients and processing cost per request.

We model the *expected response time* (under p_v) for client v to access quorum $\mathcal{Q}$ as:

$$\Delta_f(v) = \sum_{Q \in \mathcal{Q}} p_v(Q)\, \rho_f(v, Q). \tag{4.2.2}$$

The objective function we seek to minimize here is the average response time over all clients: $\mathsf{avg}_{v \in V}[\Delta_f(v)]$. If we set $\alpha = 0$, the response time for a client request becomes $\delta_f(v, Q) = \max_{w \in f(Q)} d(v, w)$. This transforms the objective function into average network delay, a measure studied previously (Fu [1997], Gupta et al. [2005], Kobayashi et al. [2001], Lin [2001]).

In our evaluation of quorum placements we make a distinction between one-to-one and many-to-one placements. Each of these two categories of placements potentially has advantages over the other: for instance, many-to-one placements can decrease network delay by putting more logical elements on a single physical node. One-to-one quorum placements, on the other hand, are important when we want to preserve the fault-tolerance of the original quorum system. We use the techniques described in Sections 2.3.1and 2.3.2 as our one-to-one placement for Grid and Majority and the algorithm from Section 2.2.3 as our many-to-one placement.

A special many-to-one quorum placement against which we evaluate our algorithms is the *singleton* quorum placement: this puts all the elements of U on a single network node (regardless of that node's capacity). The node on which we place all the universe elements is the node that minimizes the sum of the distances from all the clients to itself. When all nodes of the graph are clients, this node is also known as *the median* of the graph. Lin [2001] showed that the singleton is a 2-approximation for the problem of designing a quorum system over a network to minimize average network delay.

4.2.2 New techniques

Optimizing access strategies Our first new technique is an algorithm that, given a placement, finds client access strategies that minimize network delay under a set of node capacity constraints. The algorithm allows one to improve network delay while preserving per-server load, something that will be useful when we want to minimize response time.

The algorithm is based on a LP with variables $p_{vi} \geq 0$, where p_{vi} specifies the probability of access of quorum Q_i by client v. We assume a quorum placement $f : U \to V$ and a capacity $\mathsf{node\_cap}(v)$ for each $v \in V$ are given. A LP formulation of the problem is:

$$\text{minimize } \mathsf{avg}_{v \in V} \sum_{i=1}^{m} p_{vi} \delta_f(v, Q_i) \tag{4.2.3}$$

$$\mathsf{avg}_{v \in V} \mathsf{load}_{v,f}(v_j) \leq \mathsf{node\_cap}(v_j) \qquad \forall v_j \in V \tag{4.2.4}$$

$$\sum_{i=1}^{m} p_{vi} = 1 \qquad \forall v \in V \tag{4.2.5}$$

$$p_{vi} \in [0, 1] \qquad \forall v \in V, \forall Q_i \in \mathcal{Q} \tag{4.2.6}$$

Constraints (4.2.4) set capacity constraints for graph nodes. Constraints (4.2.5)–(4.2.6) ensure that the values $\{p_{vi}\}_{i \in \{1,\ldots,m\}}$ constitute an access strategy, i.e., a distribution on quorums. Since $p_{vi} \geq 0$ are positive reals, we can always find a solution in time polynomial in $\max(m, n)$, if one exists. A solution might not exist if, e.g., the node capacities are set too low.

An iterative algorithm The many-to-one placement algorithm from Section 2.2.3 can be combined with the access-strategy-optimizing algorithm above in an iterative way. Let $\mathsf{avg}(\{p_v\}_{v \in V})$ denote the access strategy p defined by $p(Q) = \mathsf{avg}_{v \in V} p_v(Q)$. In addition, let p_v^0 be the uniform distribution for all $v \in V$, and let $\mathsf{node\_cap}^0(v)$ be the capacity of v input to the algorithm. Iteration $j \geq 1$ of the algorithm proceeds in two phases:

1. In the first phase of iteration j, the many-to-one placement algorithm from Section 2.2.3 is executed with $\mathsf{node\_cap}(v) = \mathsf{node\_cap}^0(v)$ for each $v \in V$ and with access strategy $p = \mathsf{avg}(\{p_v^{j-1}\}_{v \in V})$, to produce a placement f^j. Recall that it is possible that for some nodes v, $\mathsf{load}_{f^j}(v) > \mathsf{node\_cap}^0(v)$, though the load can exceed the capacity only by a small factor.

2. In the second phase of iteration j, the access-strategy-optimizing algorithm above is executed with $\mathsf{node\_cap}(v) = \mathsf{load}_{f^j}(v)$ for each $v \in V$ to produce new access strategies $\{p_v^j\}_{v \in V}$.

After each iteration j, the expected response time (4.2.2) is computed based on the placement f^j and access strategies $\{p_v^j\}_{v \in V}$. If the expected response time did not decrease from that of iteration $j-1$, then the algorithm halts and returns f^{j-1} and $\{p_v^{j-1}\}_{v \in V}$.

Note that this algorithm can only improve upon the many-to-one placement algorithm of Section 2.2.3, since the second phase can only decrease average network delay while leaving loads unchanged, and because the algorithm terminates if an iteration does not improve the expected response time.

4.3 Simulation methodology

We implemented the algorithms in Section 4.2 in C and GNU MathProg (a publicly available LP modeling language). To solve the LPs we use the freely available glpsol solver. The version of glpsol we use (4.8) can solve LPs with up to 100,000 constraints, which limits the systems for which we can evaluate our algorithms.

Network topologies The network topologies that we consider come from two sources: The first is a set of ping round trip times (RTTs) measured between 50 different Planetlab sites over a 5 month period in the second half of 2006 (`http://ping.ececs.uc.edu/ping/`). We call this topology "Planetlab-50". The second dataset is built from pairwise delays measured between 161 web servers using the king latency estimation tool (Gummadi et al. [2002]). The set of web servers was obtained from a publicly available list used in previous research (`http://ww.icir.org/tbit/daxlist.txt`). We call this topology "daxlist-161".

Quorum systems We evaluate our algorithms for four quorum systems: three types of Majorities commonly used in protocol implementations (the $(t+1, 2t+1)$, $(2t+1, 3t+1)$ and $(4t+1, 5t+1)$ Majorities) and the $k \times k$ Grid. In each experiment we vary the universe size by varying the t and k parameters from 1 to the highest value for which the universe size is less than the size of the graph.

Measures Our results in the following sections were obtained by computing one of two measures: average response time, $\mathsf{avg}_{v \in V}[\Delta_f(v)]$, where $\Delta_f(v)$ is defined according to (4.2.2), or average network delay, which is computed identically but with $\alpha = 0$ in (4.2.1).

4.4 Low client demand

In this section we consider the case when client demand in the system is low. This can be modeled by setting $\alpha = 0$ in definition (4.2.1), as the response time in this case is well approximated by the network delay.

Lin [2001] showed that the singleton placement yields network delay within a factor of two of *any* quorum system placed in the network. Thus, for a system with low client demand, i.e., in which network delay is the dominant component of response time, using a quorum system cannot yield much better response time than a single server. However, a quorum system might still yield advantages in fault-tolerance, and so our goal will be to determine the performance costs one pays while retaining the fault-tolerance of a given quorum system, i.e., by placing it using a one-to-one placement.

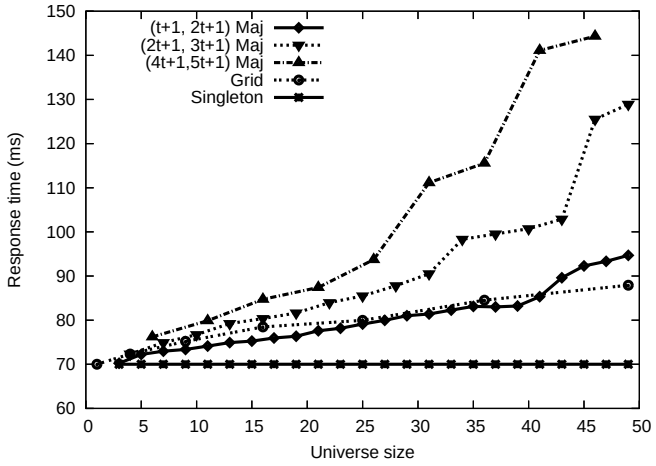


Figure 4.4.3: Response times on Planetlab-50; $\alpha = 0$; ClosestDly access strategy

Since we are trying to minimize network delay, clients should use for each access the closest quorum (or quorums if more than one have the same smallest distance to v), i.e., $p_v(Q_i) = 1/nr\_clo$ for all the nr_clo v's closest quorums Q_i, and $p_v(Q') = 0$ for all others; we call this the ClosestDly quorum access strategy.

The results of this analysis on the Planetlab-50 topology are shown in Figure 4.4.3. These results suggest that the average response time increases for each quorum placement as the universe size grows. In some cases the increase exhibits a critical point phenomenon: response time degrades gracefully up to a point and then degrades quickly. This can be best seen for the $(2t + 1, 3t + 1)$ and $(4t + 1, 5t + 1)$ Majorities.

A second observation is that for a fixed universe size, the response time is better for

quorum systems with smaller quorums. In almost all the graphs the line corresponding to Grid is the best after the singleton, the $(t+1, 2t+1)$ Majority is the second, etc. More surprisingly, the response times for the quorum systems with small quorum sizes are not much worse than that of one server up to a fairly large universe size. The exact values depend on the topology; more generally, from other experiments we performed it seems that the values depend on the distribution of average distances from nodes of the graph to all clients.

In conclusion, under low client demand, using quorum systems with smaller quorum sizes gives better performance. For all quorum systems, the degradation in performance as compared to one server is fairly small up to a certain universe size that depends on the topology and the particular quorum system considered.

4.5 High client demand

In this section we evaluate algorithms for minimizing response time when there is high client demand in the system. We start by looking at one-to-one placements when clients use either the ClosestDly access strategy from Section 4.4 or a Balanced strategy in which p_v is the uniform distribution for each client v.

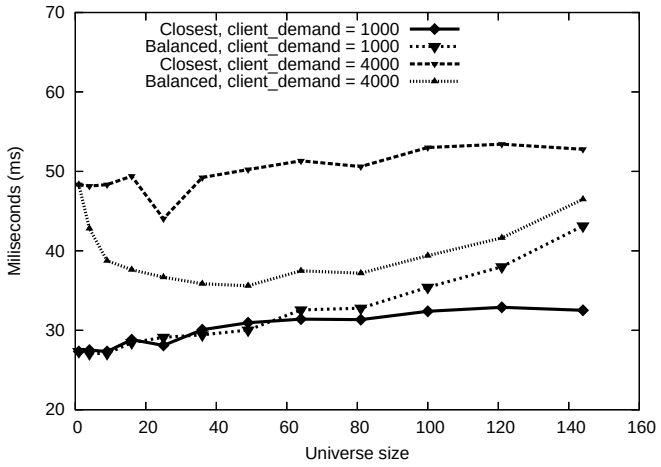


Figure 4.5.4: Response time for Grid for different client demands on daxlist-161

To compute response time we set the α parameter as follows: $\alpha = op\_srv\_time * client\_demand$. We use a value of .007 ms per request for the op_srv_time parameter (this is the time needed by a server to execute a Q/U write operation on a Intel 2.8GHz

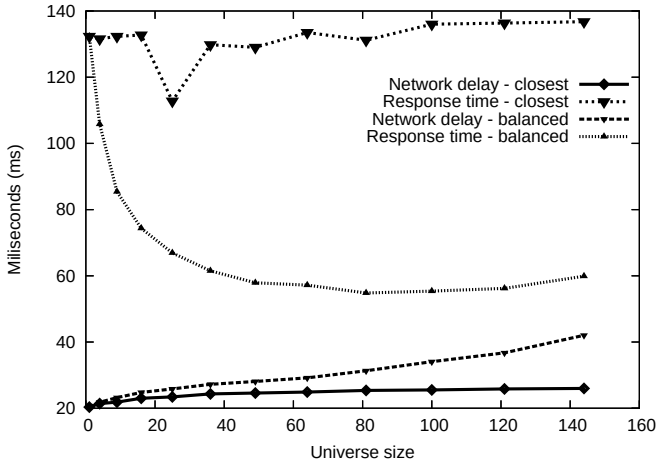


Figure 4.5.5: Grid with $client\_demand = 16000$ on daxlist-161

P4). We set $client\_demand$ to either 1000, 4000 or 16000 requests.

In Figure 4.5.4 we plot response times for the ClosestDly and Balanced strategies for the Grid quorum system when placed on the daxlist-161 topology and $client\_demand \in \{1000, 4000\}$. The results show that for low client demand, ClosestDly seems to be the best access strategy in most cases (particularly for larger universe sizes where network delays are larger, as well), while Balanced is the best access strategy for sufficiently high client demand.

Another interesting aspect illustrated by Figure 4.5.4 is the effect on response time obtained by varying the universe size (the line corresponding to Balanced for a $client\_demand$ of 4000). For small universe sizes, the processing is spread on just a few nodes, which, in the case of high client demand, has a negative impact on the response time. At the same time, for large universe sizes, each node sees a much smaller load, but now network delay is sufficiently large to become the dominating factor in client response time.

To better illustrate the effect that load balancing has on response time, we also plot results for a higher value of client demand, $client\_demand = 16000$ in Figure 4.5.5. We plot both response time and network delay on the same graph. The network delay component increases with the universe size, while the load component decreases for an access strategy that balances load on servers. Since the load induced by client demand in this case is significantly larger than the network delay component, the response time for the Balanced access strategy actually decreases with increased universe size. At the same time, the response time of the ClosestDly access strategy does not exhibit this behavior, since this provides no load balancing guarantees on the nodes.

In conclusion, while ClosestDly is the best access strategy for sufficiently low client demand (Section 4.4), Balanced is the best for very large client demand. There is also a region of client demand values for which none of the two access strategies performs better than the other; this is clearly visible in Figure 4.5.4 where the lines of the two access strategies for $client\_demand = 1000$ cross each other in multiple points. Below we present a technique for finding access strategies to minimize the response time for an arbitrary client demand.

4.6 Moderate client demand

To find the best access strategy for a given topology, quorum system, quorum placement and client demand in the region where none of ClosestDly and Balanced is best, we will use LP (4.2.3)–(4.2.6) with different values for the capacity of nodes. While in practice the capacity of a machine is determined by its physical characteristics, here we use $\mathsf{node\_cap}(v)$ as a parameter to manipulate the clients' access strategies so as to minimize response time. To use this technique in the real world, we can simply determine an upper bound for $\mathsf{node\_cap}(v)$ of a machine v based on its physical characteristics and then run this tool with $\mathsf{node\_cap}(v)$ no higher than the obtained upper bound.

We evaluate this technique in the following way: we choose a set of 10 values c_i in the interval $[L_{opt}, 1]$ and set the node capacity of all nodes, $\mathsf{node\_cap}(v) = c_i$, for each $i \in \{1, \ldots, 10\}$. L_{opt} here is the optimal load of the quorum system considered, for a fixed universe size. We solve LP (4.2.3)–(4.2.6) for each value c_i to obtain a set of access strategies (one for each client) and then compute the response time corresponding to each such set of access strategies. Finally, we pick the value c_i that minimizes the response time. In our evaluation we choose the values c_i to be:

$$c_i = L_{opt} + i \cdot \lambda \tag{4.6.7}$$

for $i \in \{1, \ldots 10\}$, where $\lambda = (1 - L_{opt})/10$.

Figure 4.6.6 shows how the response time changes when we vary the node capacities for different universe sizes, assuming a client demand of $client\_demand = 16000$. In general, setting a higher node capacity allows clients to access closer quorums, thus decreasing network delay but increasing the load component at some of the nodes at the same time. For high client demand, this can yield worse response times, since nodes with a high capacity will become the bottleneck; i.e., the costs of high load will outweigh the gains in network delay. In this case it makes sense to disperse load across as many nodes as possible, which can be enforced by setting low node capacities.

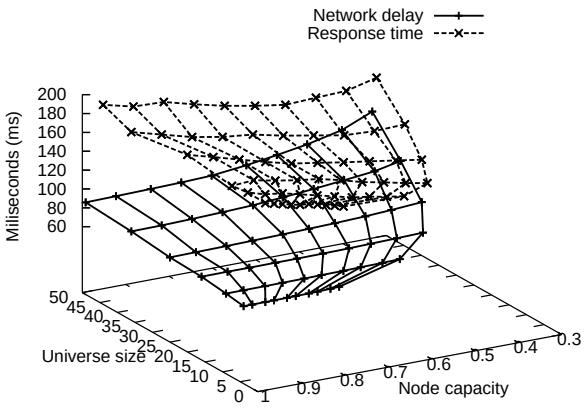


Figure 4.6.6: Grid when increasing node capacities on Planetlab-50

Non-uniform node capacities A variation of the previous technique can help improve the response time further. This approach is based on the following observation: for a given c_i, LP (4.2.3)–(4.2.6) will find access strategies that minimize network delay by selecting quorums that are close to clients, as much as the capacity of graph nodes permits. As a result some nodes will have their capacity saturated, and thus will handle the same volume of requests, irrespective of their average distance to the clients. For this set of nodes, the response time will thus depend on their average distance to the clients: for server nodes further away clients will have to wait more than for closer nodes.

This observation leads us to following natural heuristic: we can set nodes capacities inversely proportional to their average network delay to the clients. This will hopefully spread load across servers in a way that minimizes response time.

We now present in more detail the algorithm for setting node capacities. Let $\{v_1, \ldots, v_n\}$ be the support set of a given placement (we assume only one-to-one placements here), and let s_i be the average distance from all clients to v_i. Our goal is to set capacity $\mathsf{node\_cap}(v_i)$ to be inversely proportional to s_i and in a given range $[\beta, \gamma] \subseteq [0, 1]$. Let $le = \min_{i \in 1..n} \frac{1}{s_i}$ and $re = \max_{i \in 1..n} \frac{1}{s_i}$. We then assign

$$\mathsf{node\_cap}(v_i) = \frac{1/s_i - le}{re - le} (\gamma - \beta) + \beta$$

So, for example, if $v_i = \arg\min_{i \in 1..n} \frac{1}{s_i}$, then $\mathsf{node\_cap}(v_i) = \beta$, and if $v_i = \arg\max_{i \in 1..n} \frac{1}{s_i}$, then $\mathsf{node\_cap}(v_i) = \gamma$.

We evaluate this method for the Grid quorum system with universe size ranging from

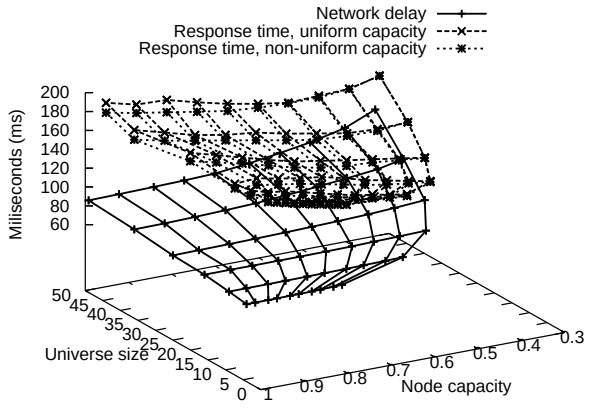


Figure 4.6.7: Grid on Planetlab-50 with uniform and non-uniform node capacities

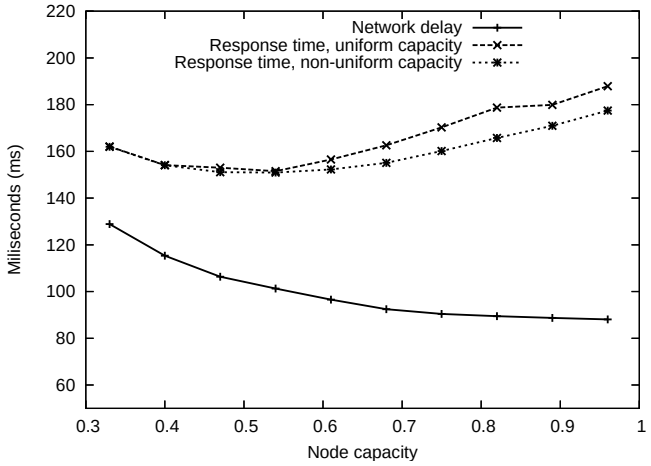


Figure 4.6.8: 7×7 Grid on Planetlab-50 with uniform and non-uniform node capacities

4 to 49 on the Planetlab-50 topology. To compare with the results for uniform node capacities above, we use intervals $[\beta, \gamma] = [L_{opt}, c_i]$ for $i = 1..10$ (see (4.6.7)). We set $client\_demand = 16000$ for this set of experiments.

Figure 4.6.7 shows response times for both uniform and non-uniform node capacities. For small capacities the two approaches give almost identical results. As capacities increase, the heuristic for non-uniform capacities gives better response time than the algorithm for uniform node capacities. The reason: for small values of c_i, the length of the $[\beta, \gamma]$ interval is close to 0, and as such, the nodes from the support set have almost the

same capacity. As the $[\beta, \gamma]$ interval grows, the capacities are better spread out and better (inverse proportionally) match the distances s_i. This spreads load over nodes with larger average distances to the clients, which decreases response time.

To see the improvement given by this heuristic we also plot results for a fixed universe size ($n = 49$). Figure 4.6.8 shows that increasing node capacity increases the response time as well (due to the high load in the system) but at a slower pace for our heuristic than for the algorithm with fixed node capacities. As the size of client demand increases, we expect this effect to become more pronounced.

Evaluation of the iterative approach So far we have evaluated only algorithms yielding one-to-one placements. The last technique for improving response time that we evaluate in this paper is the iterative approach described in Section 4.2.2. Since this approach creates many-to-one placements, network delay will necessarily decrease: some of the quorums become much smaller, thereby allowing clients to reduce the distance they need to travel to contact a quorum.

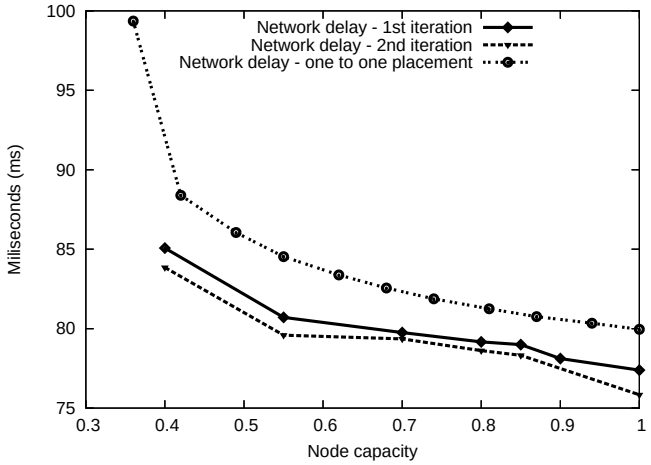


Figure 4.6.9: Network delay for iterative approach for 5 x 5 Grid on Planetlab-50

In Figure 4.6.9 we show the performance gains in terms of network delay compared to a one-to-one placement for a 5×5 Grid. We run the iterative algorithm for different values of the node capacity to see whether the improvement in network delay depends on node capacity. For all node capacities the best improvement comes after the first phase, at the end of which many universe elements are placed on the same node. The second phase brings only small improvements. Most of the runs terminate after the first iteration.

We have also evaluated the response time for each intermediary point in this iterative

process. The results show that using many-to-one placements can increase or decrease response time depending on the placement found in the first phase of the first iteration and on the client demand. For instance, if the placement found puts multiple quorum elements on many nodes of the graph, the response time increases with the client demand. For low client demand, response time is usually better than for the one-to-one placements. Finally, the response time is always improved between the first and the second phases of the first iteration, but only by small values (usually between 2 and 5 ms). Consequently, using many-to-one placements improves response time over other approaches mostly in the case of low client demand. However, the techniques discussed in Section 4.4 also excel in this case, while retaining the fault-tolerance of the original quorum system.

4.7 Related work

The earliest empirical study of which we are aware of quorum behavior in wide-area networks is due to Amir and Wool [1996]. Their analysis studied the availability of quorums deployed across three wide-area locations, with a focus on how the behavior of the wide-area network violated typical assumptions underlying theoretical availability analyses of quorum systems. Their focus on availability is complementary to ours on response time, and was conducted on much smaller topologies than what we consider.

Bakr and Keidar [2002] conducted a wide-area study of a *communication round* among nodes, in which each node sends information to each other participating node. Their study also focused on delay, though otherwise their study and ours are complementary. Our study considers only direct client-to-quorum communication; their treatment of four different round protocols is more exhaustive. However, their study is confined to one node at each of ten wide-area sites—they do not consider altering the number or locations of nodes—and does not admit load dispersion (since all nodes participate in all exchanges). In contrast, our study shows the impact of scaling the number of servers in a wide-area quorum system; of the judicious placement of servers among the candidate sites; and of tuning client access strategies to disperse load.

Amir et al. [2006] construct and evaluate a Byzantine fault-tolerant service with the goal of improving the performance of such a service over a wide-area network. In this context, they evaluate BFT (Castro and Liskov [2002]) and their alternative protocol, which executes Paxos (Lamport [1998]) over the wide area. These protocols can be viewed as employing Majority (Gifford [1979], Thomas [1979]) quorum systems in which a quorum constitutes greater than two-thirds or one-half of the wide-area sites, respectively; their evaluation overlaps ours in that we also evaluate these quorum systems (and others). As in

the Bakr and Keidar evaluation, however, Amir et al. constrain their evaluation to a fixed number of wide-area sites (in their case, five), and do not consider altering the number or locations of nodes. Consequently, our evaluation is complementary to that of Amir et al. in the same ways.

Oppenheimer et al. [2005] examined the problem of mapping large-scale services to available node resources in a wide-area network. Through measurements of several services running on Planetlab (Bavier et al. [2004]), they extracted application resource demand patterns and CPU and network resource availability for Planetlab nodes over a 6-month period. From these measurements, they concluded that several applications would benefit from a service providing resource-informed placement of applications to nodes. Among other interesting results, their study reveals that inter-node latency is fairly stable and is a good predictor of available bandwidth, a conclusion that supports our focus on periods in which latencies between nodes are stable.

4.8 Conclusions

In this chapter we have evaluated techniques for placing quorum systems on a wide-area network to minimize average client response time. The results of this evaluation reveal several interesting facts. First, for low client demand, using quorum systems up to a limited universe size in certain topologies does not substantially degrade performance compared to a single node solution. Thus, quorum systems are a viable alternative to the singleton solution in such cases, and offer better fault-tolerance. Second, as the client demand increases, it is important to balance load across servers to obtain good response time. When the network delay and client demand both play important roles in the response time, finding the right strategies by which clients access quorums is crucial to minimizing response time. Our methods for optimizing clients' access strategies, used with both uniform and non-uniform node capacities, are especially useful here.

Finally, in our current framework, using many-to-one (instead of one-to-one) placements improves the response time only for low values of client demand. A variation of our model, in which a server hosting multiple universe elements would execute a request only once for all elements it hosts, can clearly improve the performance. We plan to analyze the benefits of such an approach in future work.

Chapter 5

Dynamic Quorum Selection on Wide-Area Networks

In this chapter we continue our efforts to improve the performance of quorum-based protocols in wide-area networks (WANs) by looking at the problem of adapting the clients access strategies in response to workload changes. This complements the work from the previous chapter where we developed quorum selection algorithms for static workloads.

Workloads can change substantially over different times of the day due, e.g., to diurnal patterns of activity or flash-crowd events (Jung et al. [2002], Padmanabhan and Sripanidkulchai [2002], Ranjan et al. [2002]). Here we show that the response times achievable by clients can be improved substantially if the quorums they access are chosen so as to account for the overall client activity in the system at that time and the focal points of that activity — i.e., the clients that are inducing load or, more to the point, the servers that are being subjected to it. Intuitively, a client should select a nearby quorum for its operation if those servers are lightly loaded (so as to minimize network delays), but should choose a less lightly loaded quorum, albeit potentially further away, if nearby quorums are too busy. To our knowledge, the problem of designing load-dispersing algorithms for quorum systems on WANs for balancing this tension dynamically has not been considered before (see Section 5.4 for a discussion of related work).

To address this problem we propose two quorum selection algorithms for dynamic workloads. The algorithms enable each client to independently choose the quorum it accesses for each operation it invokes. The client does so with no communication with other clients and with only the information it gleans from previous operation invocations to the service. As such, these algorithms impose negligible overheads on servers. We show, however, that our algorithms quickly adapt clients' choices of quorums in the face of workload

dynamics, and that the clients' quorum choices that result minimize client response times and maximize client throughput nearly optimally across a broad range of workloads. We draw these conclusions based upon evaluations of our algorithms on a wide-area topology derived from PlanetLab (Bavier et al. [2004]) measurements, emulated using Modelnet (Vahdat et al. [2002]), and using two quorum-based operation-invocation protocols: a generic Grid quorum (Cheung et al. [1992], Kumar et al. [1993]) protocol and the Q/U (Abd-El-Malek et al. [2005]) protocol using a Majority quorum system.

The rest of this chapter is structured as follows. We describe the terminology and evaluation methodology for the algorithms we propose in Section 5.1. We describe these algorithms and evaluate them in Section 5.2 and Section 5.3. We discuss related work in Section 5.4 and conclude in Section 5.5.

5.1 Preliminaries

5.1.1 Terminology and assumptions

We start with a brief recollection of the previously defined notations for a quorum system $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ over a universe U and for the access strategy p_c employed by a client c (for all clients c, $\sum_{Q_i \in \mathcal{Q}} p_c(Q_i) = 1$).

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 (e.g., Gummadi et al. [2002], Ng and Zhang [2002]). Clients also need a mechanism for estimating the load at servers. Rather than modeling this explicitly we choose to derive this information from the response time observed by clients when accessing servers.

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 $Q \in \mathcal{Q}$ (this is the max-delay cost function defined in Chapter 2).

Two access strategies are known to perform well for certain workloads on wide area networks. One, denoted ClosestDly, is to always access the closest quorum in terms of network latency, i.e., 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. 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. That is, the response times are overwhelmingly dominated by network delays, and so it makes

sense for each client to choose from among their closest quorums. The second, denoted 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. (This is the case for all quorum systems we consider here.) That is, for each client c, 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. That is, networking costs are so inconsequential that it is not necessary to take them into account in quorum selection.

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.

5.1.2 Evaluation setup

For evaluation purposes we consider two specific quorum constructions: the Majority (Gifford [1979], Thomas [1979]) and the Grid (Cheung et al. [1992], Kumar et al. [1993]). In particular we performed experiments with Q/U (Abd-El-Malek et al. [2005]), a Byzantine fault-tolerant quorum system protocol that uses the $(4b+1, 5b+1)$ Majority, i.e., in which a quorum is any $4b + 1$ out of a total of $5b + 1$ servers, and with a generic quorum system protocol we implemented that uses a $k \times k$ Grid.

We use two Planetlab topologies in our evaluation: the first is the "Planetlab-50" topology previously described in Section 4.3. For this topology 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. 93% of all pairs of nodes have a latency of at least 20 milliseconds (ms) between them. We call this topology PlanetLab50a. The second topology is also of size 50 but with an average round-trip time of 117 ms between any pair of nodes and a standard deviation of 76 ms. A fraction of 91% of all pairs are at distance of at least 20 ms apart. We call the second topology PlanetLab50b. We deploy the two topologies on a Modelnet (Vahdat et al. [2002]) testbed.

In any given experiment, we place the servers on the nodes of a topology in a one-to-one fashion (i.e., any node hosted at most one server) using the one-to-one placement algorithms introduced in Chapter 2 and evaluated in Chapter 4. These placement algorithms minimize the average, over all nodes, of each node c's expected network delay when accessing quorums using 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., a new request is sent as soon as the previous one completed. Clients only issue requests that complete in a single round trip. While for arbitrary quorum protocols not all operations are guaranteed to complete in a single round trip, the Q/U protocol (Abd-El-Malek et al. [2005]) is designed so that most do in common cases. We set the application processing delay per client request at each server to 1 ms.

We evaluate our algorithms with respect to several measures. *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 throughput of all clients. *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 measured the time it took for clients to transform their access strategies into ClosestDly. We say that the algorithm has converged to ClosestDly when the L2 norm between ClosestDly 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 clients to change their access strategies to Balanced. Finally, we also measure the *overhead* associated with each algorithm, where by overhead we understand the average time per request spent in the quorum-selection function.

5.2 A weight shifting algorithm

A *dynamic quorum selection algorithm* should change the access strategies used by clients in response to changes in server loads. Thus we need a mechanism for detecting when such changes take place. Ideally this mechanism should be able to detect other events that may impact performance as well, such as network congestion, for instance.

There are two ways of designing a load-monitoring module: servers can explicitly maintain usage information and provide it to clients using a push or a pull-based model, or clients can infer the level of load experienced by servers from the response times of their requests. (Clearly the response time of a request increases with the load on servers.) A push-based model is not very scalable, since servers need to know about all clients that might need load information. A pull-based model seems better in that respect, but one still has to deal with the overhead created by the clients' requests for load information. This motivates our choice of having clients monitor load from the variations in response times

of their requests.

Although this is the most inexpensive mechanism, it hides the following trade-off: since the answer to a client request comes from a single quorum of servers, the client will not always have the most up-to-date load information about *all* servers. Yet to make an informed decision when choosing the quorum for its next access, a client needs this information for all servers. We solve this problem by maintaining *estimates* of response times of all servers based on the recent history of requests by this client. The way in which our algorithms maintain these estimates differs in the two algorithms we present below.

In our first algorithm, each client keeps response times it observed for each server s in the last $history\_duration$ seconds, along with timestamps 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 $rt_c(s) = \sum_{i=1}^{t} w(ts_c^i(s), ts) rt_c^i(s)$, i.e., as a non-uniform average of the response times seen in the past from this server. The values $w(ts_c^i(s), ts)$ are non-negative reals and satisfy $\sum_{i=1}^{t} w(ts_c^i(s), ts) = 1$. 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 $rt_c(s)$ to be its (round-trip) network latency to s as an (optimistic) estimate of its response time.

Obviously the accuracy of this estimation method depends on the $history\_duration$ parameter. For example, setting it too close to zero 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 need to use a sufficiently large value for it ($history\_duration$ = 1s performed well in our experiments).

Using the 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 response time. We call this algorithm ClosestRT. We evaluated ClosestRT against ClosestDly and Balanced on the 5×5 Grid ranging the number of clients on each host from 1 to 15. Figure 5.2.1 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 loose the load balancing benefits of Balanced.

ClosestRT can be seen as a Nash strategy: clients always greedily select the best quorum given the perceived system conditions. In doing so they shift all their load to a single quorum for a certain amount of time. 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

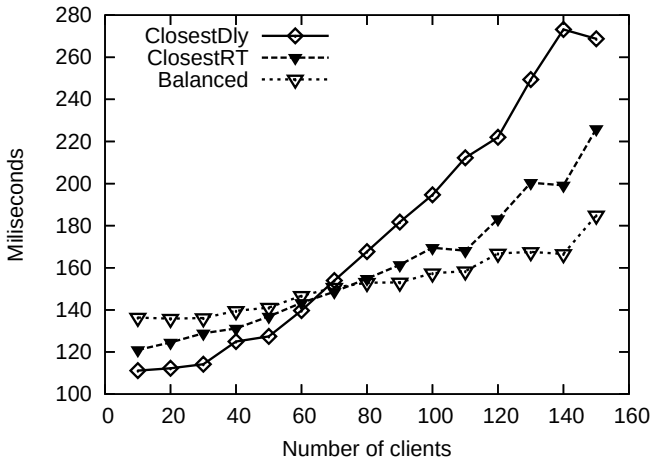


Figure 5.2.1: Response time for ClosestRT (PlanetLab50a)

strategy. This leads us to the following simple algorithm 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 $p_c(Q_j) = 0$. Thus when choosing the quorum from which load is shifted, the client must choose a quorum with positive weight. 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 would like to select a quorum with nonzero weight that contains as many servers with the high response times as possible. This can be done by selecting the quorum for which the response times of its servers, ordered in decreasing order, is lexicographically the largest among all positive-weight quorums.

We call the resulting algorithm ShiftWt. In Figure 5.2.2 we use Balanced as the initial access strategy, since it 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.

In Figures 5.2.3 and 5.2.4 we compare ShiftWt to ClosestDly, ClosestRT and Balanced for the 5×5 Grid quorum system on the PlanetLab50a topology. The results show that 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 of them.

INITIALIZATION:

for $i = 1$ to m

$p_c(Q_i) = 1/m$;

BEFORE EACH REQUEST:

for each $s \in U$, estimate $rt_c(s)$;

for each $Q \in \mathcal{Q}$, define $rt_c(Q) = \langle rt_c(s) \rangle_{s \in Q}$, sorted in decreasing order

let Q_j be such that:

i) $p_c(Q_j) > 0$ and

ii) for each $Q \in \mathcal{Q}$, if $p_c(Q) > 0$ then $rt_c(Q_j) \geq rt_c(Q)$ in lexicographic order

let Q_i be such that for each $Q \in \mathcal{Q}$, $rt_c(Q) \geq rt_c(Q_i)$ in lexicographic order;

$p_c^{\mathsf{new}}(Q_j) = p_c(Q_j)/2$;

$p_c^{\mathsf{new}}(Q_i) = p_c(Q_i) + p_c(Q_j)/2$;

$p_c^{\mathsf{new}}(Q_k) = p_c(Q_k)$, for $k \neq i, j$;

$p_c = p_c^{\mathsf{new}}$;

Figure 5.2.2: ShiftWt algorithm at client c

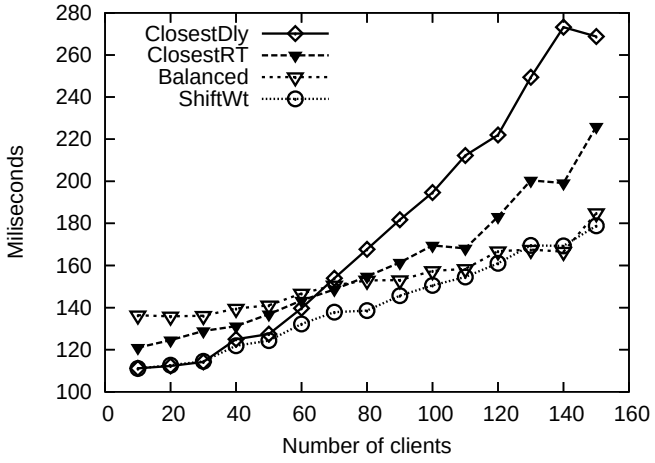


Figure 5.2.3: Response time for ShiftWt (5×5 Grid, PlanetLab50a)

Unfortunately, the convergence time of ShiftWt was less attractive in our experiments. ShiftWt converges in 48 seconds to ClosestDly in a setting with 10 clients (clientsPerNode = 1). 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.

One way to make ShiftWt converge faster is to shift more load every time the access

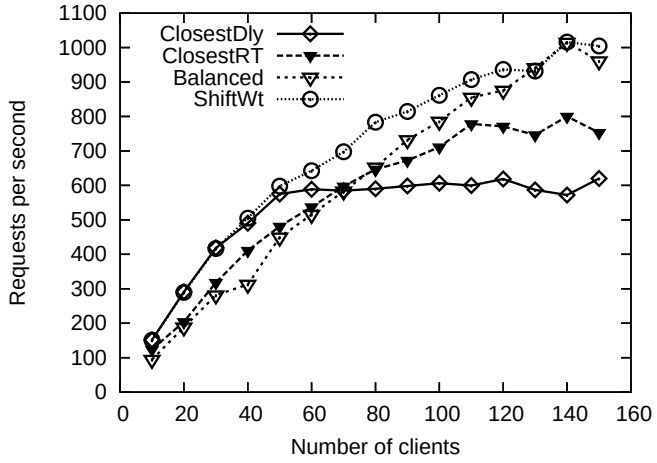


Figure 5.2.4: Throughput for ShiftWt (5×5 Grid, PlanetLab50a)

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 now is where to place all 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, so we need to do something else. Ideally what we would like is to add weight to the quorums with the smallest response times so that after absorbing this extra load, their response times are roughly the same. We use the following simple algorithm to achieve this. 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 on PlanetLab50a. Figure 5.2.5 shows convergence to ClosestDly in a low load regime while Figure 5.2.6 shows convergence to Balanced in a high load regime. (Access strategies were not altered

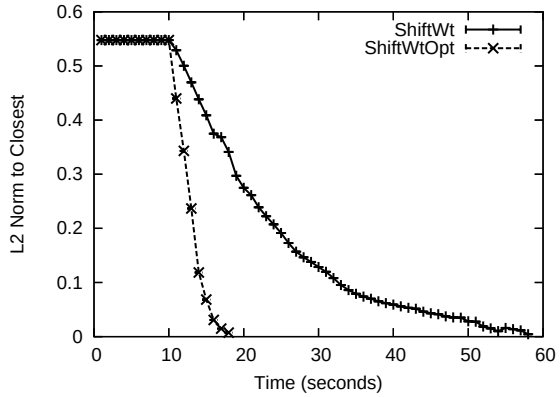


Figure 5.2.5: Convergence time to ClosestDly (5×5 Grid, PlanetLab50a)

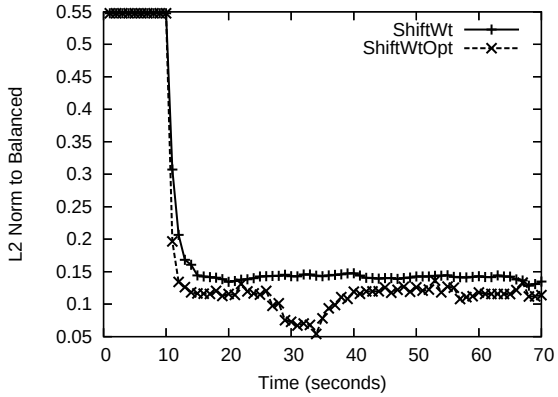


Figure 5.2.6: Convergence time to Balanced (5×5 Grid, PlanetLab50a)

b	$(4b+1, 5b+1)$ Majority		k	$k \times k$ Grid	
	ClosestDly	Balanced		ClosestDly	Balanced
1	1 s	3 s	2	5 s	3 s
2	3 s	3 s	3	6 s	1 s
3	6 s	4 s	4	7 s	2 s
4	4 s	5 s	5	8 s	2 s

Table 5.1: Convergence times for ShiftWtOpt (PlanetLab50a)

in the initial 10 seconds (s) of each run, to allow for all clients to join the experiment.) Results show a convergence time to ClosestDly of about 8 seconds (s) for ShiftWtOpt, a six-fold improvement compared to the unoptimized version, and a convergence time to Balanced of about 2 seconds for ShiftWtOpt, a two-fold improvement compared to the unoptimized version. Also, the optimized version of ShiftWt seems to get closer to Balanced that the unoptimized version.

To better understand the behavior of ShiftWtOpt, we study the convergence time and the overhead associated with it for different universe sizes for both the $(4b+1, 5b+1)$ Majority and the $k \times k$ Grid. Table 5.1 shows the convergence times to ClosestDly and Balanced for the $(4b+1, 5b+1)$ Majority for $b = 1 \ldots 4$ and for the $k \times k$ Grid with $k = 2 \ldots 5$ on PlanetLab50a. Results indicate that ShiftWtOpt converges faster to Balanced on the Grid but slower to ClosestDly than on the Majority, which seems rather unintuitive. What happened in fact is that, while for the Grid the L2 norm to ClosestDly approached 0 at convergence time, for Majority it varied between 0.1 and 0.2 (after decreasing from 0.4 or

b	$(4b+1, 5b+1)$ Majority	k	$k \times k$ Grid
1	11.39 μs	2	8.31 μs
2	44.82 μs	3	14.52 μs
3	659.24 μs	4	23.92 μs
4	41026.65 μs	5	38.03 μs

Table 5.2: Client overhead per request for ShiftWtOpt (PlanetLab50a)

0.5), especially as b became larger. By examining the data more carefully we observed that for those quorum sizes, ClosestDly assigned equal weight to a large number of "closest" quorums. This number of quorums, however, was almost never equal to the number of quorums with low response time on which we shifted weight in ShiftWtOpt and so the access strategy to which ShiftWtOpt converged was different from ClosestDly .

Table 5.2 shows the client-side overhead in microseconds (μs) of ShiftWtOpt for different universe sizes for both quorum systems considered here on PlanetLab50a. As Table 5.2 shows, the overhead increases with the number of quorums, reaching approximately 40 ms per request for $b = 4$ ($m = 5985$ 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 the increasing overhead of ShiftWtOpt as the system scales, since it has to at least examine all the quorums containing the server with the highest response time, which is a number of quorums which, for any Majority quorum system, is exponential in the number of servers.

The results from Table 5.2 suggest though 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 tens of milliseconds, an amount of time about three orders of magnitude higher than the overhead for the Grid quorum sizes we consider. Nevertheless, the rather poor convergence time and client overheads of ShiftWtOpt for Majorities provide sufficient motivation to search for a better quorum selection algorithm.

5.3 A faster converging algorithm

Our new algorithm is based on the following observations. First, in the case of low load, clients should access their closest quorums. As load increases, client response times increase as well, exceeding at some point the network delay to the clients' second-best (delay-wise) quorum (or quorums). When this happens, it makes sense to start using these

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 5.3.7: DelayBins algorithm at client c

quorums as well, since one can hope that, by doing so, load is shifted from closer to more distant but less-loaded servers, thereby decreasing response time.

Here is a more complete description of the algorithm. 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 $d_c^1 \leq \ldots d_c^t$ 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, $H_c^j = G_c^1 \cup \ldots G_c^j$. 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 5.3.7.

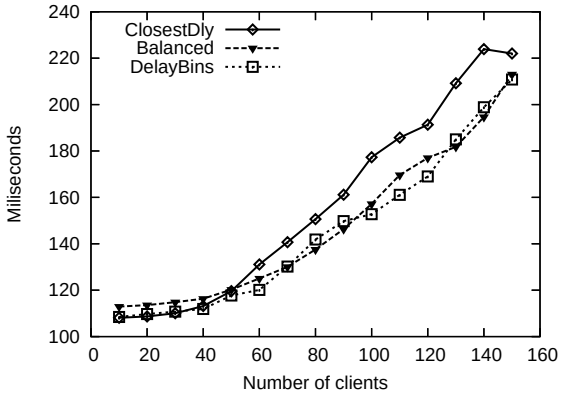


Figure 5.3.8: Response time ((4b+1,5b+1) Majority, PlanetLab50a)

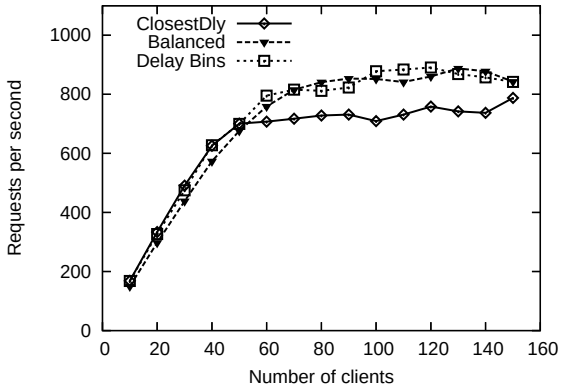


Figure 5.3.9: Throughput ((4b+1,5b+1) Majority, PlanetLab50a)

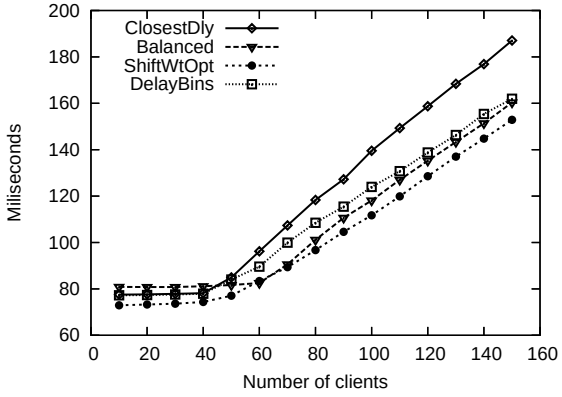


Figure 5.3.10: Response time for DelayBins ((4b + 1,5b + 1) Majority, PlanetLab50b)

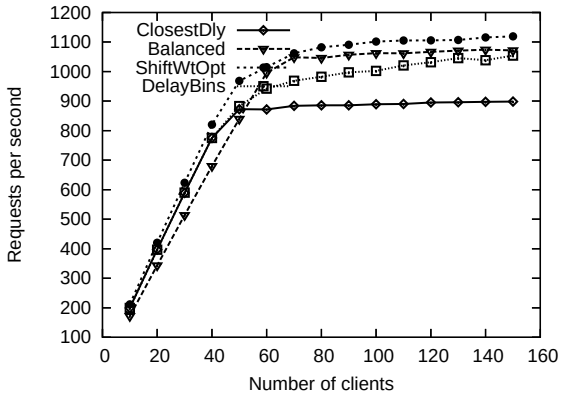


Figure 5.3.11: Throughput for DelayBins ((4b + 1,5b + 1) Majority, PlanetLab50b)

We compare the performance of DelayBins to that of ClosestDly and Balanced for different client demands on both PlanetLab50a and PlanetLab50b. Figure 5.3.8 and Figure 5.3.9 show the response time and throughput achieved by DelayBins for the $(4b{+}1, 5b{+}$

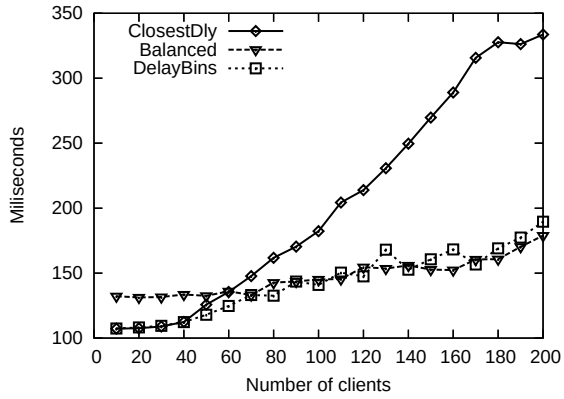


Figure 5.3.12: Response time (5×5 Grid, PlanetLab50a)

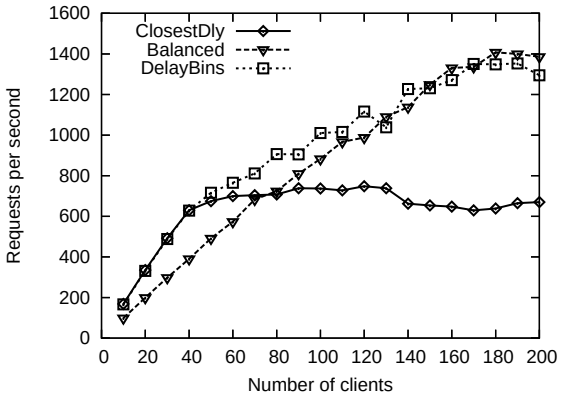


Figure 5.3.13: Throughput (5×5 Grid, PlanetLab50a)

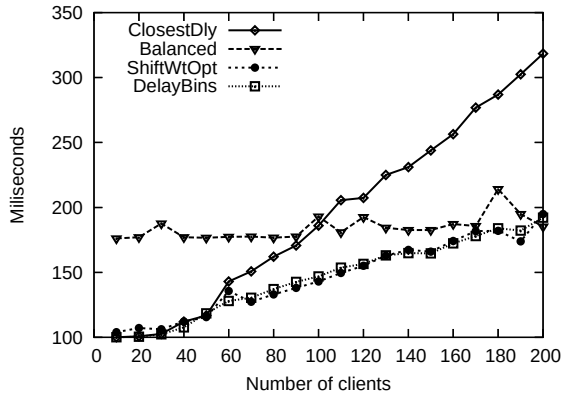


Figure 5.3.14: Response time (5×5 Grid, PlanetLab50b)

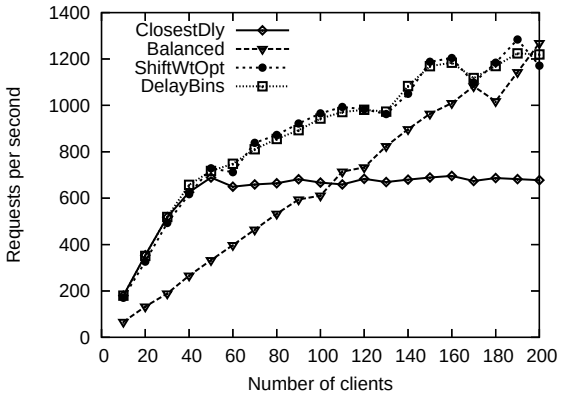


Figure 5.3.15: Throughput (5×5 Grid, PlanetLab50b)

1) Majority with $b = 2$ on PlanetLab50a. In both low-load and high-load cases, DelayBins has the same performance as the best access strategies for those cases (ClosestDly and Balanced, respectively). A similar conclusion can be drawn from our experiments on PlanetLab50b, the results of which are presented in Figure 5.3.10 and Figure 5.3.11. For a head to head comparison we also include in these figures results for the ShiftWtOpt algorithm. Figure 5.3.10 and Figure 5.3.11 show that in high and low load regimes ShiftWtOpt and DelayBins perform similarly with respect to the response time and throughput measures, but in a moderate load regime ShiftWtOpt seems to slightly outperform DelayBins, at least for this Majority quorum system. An explanation for this behavior is the rigidity of the DelayBins algorithm: at any point in time DelayBins is a Balanced access strategy

that spreads load evenly on a certain set of servers without paying attention to the delays to these servers. On the other hand ShiftWtOpt is more flexible in that it spreads load inversely proportional to the quorum delays thereby shifting load from far away quorums to closer quorums.

In Figure 5.3.12 and Figure 5.3.13 we show the response time and throughput of DelayBins for the $k \times k$ Grid for $k = 5$ on PlanetLab50a. DelayBins yields access strategies with performance similar to the best access strategies for each value of client demand. As expected, though, the gap between ClosestDly and Balanced is larger than in the case of the $(4b+1, 5b+1)$ Majority, due to the good load dispersing properties of the Grid quorum system. Similar conclusions can be drawn from the results for the PlanetLab50b topology in Figure 5.3.14 and Figure 5.3.15.

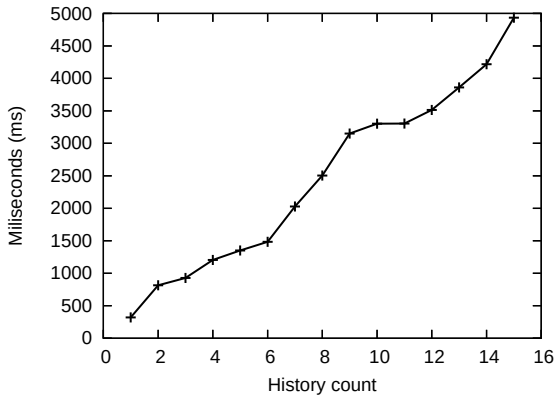


Figure 5.3.16: Convergence time dependency on $history\_count$ (PlanetLab50a)

In order to evaluate convergence time, we begin by showing the dependence of the convergence time on $history\_count$, the size of the history used for computing the average response time avg_rt for the current group. Figure 5.3.16 shows the convergence time to ClosestDly after initializing the algorithm to Balanced for different values of the history size. As expected, convergence time increases with history size. A small value of history size different from 1 seems to work best, since it gracefully reconciles the two contradictory goals of having a quickly adapting algorithm and one that does not react to sporadic spikes in load. We use a value of $history\_count = 3$ in the following experiments.

We compare the convergence time of DelayBins to that of ShiftWt and ShiftWtOpt. We perform the same experiments as before: we start clients with ClosestDly as the initial access strategy (by setting $curr\_quorum\_group = 1$, i.e., clients use the closest quorums delay-wise) 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

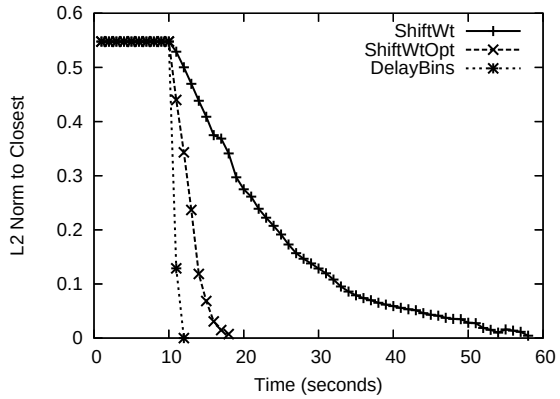


Figure 5.3.17: Convergence time to ClosestDly (5×5 Grid, PlanetLab50a)

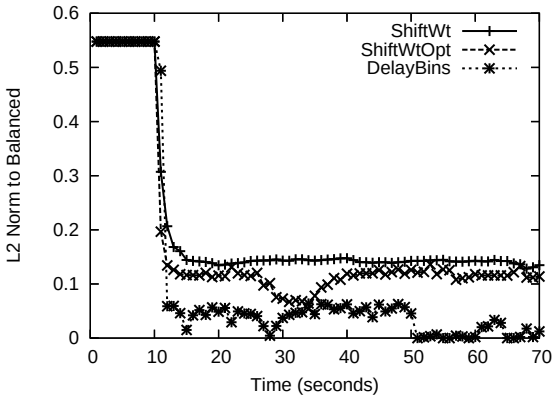


Figure 5.3.18: Convergence time to Balanced (5×5 Grid, PlanetLab50a)

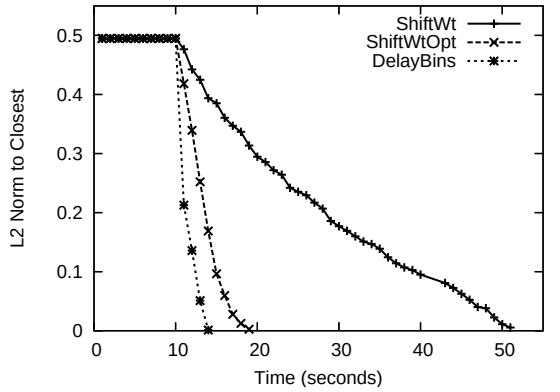


Figure 5.3.19: Convergence time to ClosestDly (5×5 Grid, PlanetLab50b)

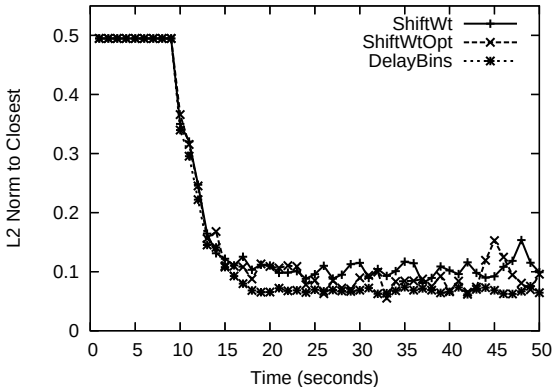


Figure 5.3.20: Convergence time to Balanced (5×5 Grid, PlanetLab50b)

also perform the converse experiment, starting from Balanced (i.e., make clients use the highest group number) and measuring how fast their average access strategy converges to ClosestDly with only 10 clients (clientsPerNode $= 1$). Figure 5.3.17 shows that DelayBins converges much faster than ShiftWtOpt to ClosestDly (in about 1–2 seconds). At the same time, Figure 5.3.18 shows that the convergence time to Balanced is about the same as that of ShiftWtOpt (again 1–2 seconds). The results for the PlanetLab50b illustrate a similar picture, with two small exceptions: Figure 5.3.19 shows that DelayBins has a slightly larger convergence time to ClosestDly than before (of about 4 seconds). The convergence to Balanced has also increased to about 5 seconds.

To perform a more exhaustive comparison we evaluate DelayBins with respect to the

b	Conv. to ClosestDly		Conv. to Balanced	
	ShiftWtOpt	DelayBins	ShiftWtOpt	DelayBins
1	1 s	1 s	3 s	2 s
2	1 s	1 s	3 s	1 s
3	6 s	4 s	4 s	2 s
4	4 s	4 s	5 s	2 s

Table 5.3: Convergence times ($(4b+1$,$5b+1)$ Majority, PlanetLab50a)

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 5.4: Convergence times ($k \times k$ Grid, PlanetLab50a)

convergence time measure for different universe sizes for the two quorum systems considered here on PlanetLab50a. Table 5.3 compares the convergence times of DelayBins and ShiftWtOpt to both ClosestDly and Balanced on the $(4b+1, 5b+1)$ Majority. In all cases, DelayBins converges as fast or faster than ShiftWtOpt. This benefit of DelayBins is further pronounced with the Grid quorum system, as can be seen in Table 5.4. 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 than DelayBins does.

The last measure with respect to which we evaluate DelayBins is client overhead. Table 5.5 lists results for both DelayBins and ShiftWtOpt for both the Majority and the Grid on PlanetLab50a. 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.

b	$(4b+1, 5b+1)$ Majority		k	$k \times k$ Grid	
	ShiftWtOpt	DelayBins		ShiftWtOpt	DelayBins
1	11.39 μs	1.32 μs	2	8.31 μs	1.64 μs
2	44.82 μs	1.51 μs	3	14.52 μs	1.69 μs
3	659.24 μs	1.84 μs	4	23.92 μs	1.82 μs
4	41026.65 μs	2.31 μs	5	38.03 μs	1.92 μs

Table 5.5: Client overhead per request (PlanetLab50a)

The primary reason for the superior client-side overhead of DelayBins is the following observation. Even though the pseudocode of Figure 5.3.7 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. As such, a performance of DelayBins can be achieved that is independent of m in practice. In fact, the execution time of DelayBins depends only 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, for large quorum system sizes the overhead associated with updating the access strategy can delay the sampling task.

5.4 Related work

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 (e.g., Carter and Crovella [1997], Guyton and Schwartz [1995], Ratnasamy et al. [2002]) or only on load at servers (Starobinski and Wu [2005]), to using both network delay and load at servers when choosing the best server (Ranjan et al. [2004]). From this perspective our work is more closely related to the last approach since we use both network delay and server load in deciding how to access quorums. On the other hand, our quorum selection algorithms are more efficient, imposing practically no overhead on servers (no external load monitoring mechanism is necessary) and only negligible overhead on clients, and make no assumptions about client behavior to achieve good performance.

The design of the response time estimation algorithm from Section 5.2 is based on the idea of placing more weight on the response times observed from more recent requests. Similar ideas have been used elsewhere (e.g., Yoshikawa et al. [1997]). More generally, there has been a lot of work on finding server selection policies that make use of possi-

bly stale load information (e.g., Mirchandaney et al. [1989, 1990], Mitzenmacher [1997], Shivaratri et al. [1992], Dahlin [1999]). In particular Mitzenmacher [1997] shows how to avoid the 'herd effect' by choosing the lightest loaded server not from all the servers, but from a randomly selected subset of size k. Dahlin [1999] extends these ideas even further by examining algorithms that interpret load information based on the age of that information. The algorithms from Dahlin [1999] also rely on certain assumptions about client interarrival time. As mentioned before, we also estimate the response time of a client access to a server by taking into account the age of *response times* from previous invocations. At the same time we make no assumptions about knowing the client interarrival time.

The idea of improving client response time by shifting load between quorums with a certain probability is based on earlier work on selfish load balancing by Berenbrink et al. [2006]. That work studies the problem of assigning each of m tasks to n resources in a way that optimizes the completion time of each task. Berenbrink et al. [2006] presents a probabilistic task reassignment policy that converges in $O(\log \log m)$ to Nash equilibrium. An important assumption made there is that the set of m tasks does not change over time. As noted previously though, workloads tend to change substantially over different times of day. This motivates our evaluation of a different convergence measure that more closely illustrates how our algorithms adapt to load changes.

Balancing load for quorum systems in general (i.e., without accounting for network delays) is a fairly well-studied problem as well. Naor and Wool [1998] defined a formal notion of load and developed load-optimal quorum constructions and access strategies to achieve optimal load for these constructions. Malkhi and Reiter [1998], Malkhi et al. [2000] extended 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 (Abd-El-Malek et al. [2005]), which we use in our evaluations, is designed for this fault model. Holzman et al. [1997] examined 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 showed methods for determining the access strategies for which a quorum system achieves optimal balancing. None of these works however shows how to adapt access strategies to dynamic workloads on WANs.

Prior work (Gupta et al. [2005], Golovin et al. [2006], Oprea and Reiter [2007], Amir et al. [2006], Amir and Wool [1996]) studies ways to optimize performance of quorum systems on WANs. Of these, only our own (Oprea and Reiter [2007]) considers the problem of finding optimal access strategies, but only for static workloads. Adapting to dynamic workloads was the focus of this chapter.

5.5 Discussion and conclusion

In this chapter we described two algorithms (ShiftWt/ShiftWtOpt, and DelayBins) for choosing quorums so as to achieve good response time and throughput for operations in wide-area networks, and that adapt quickly to workload changes. We showed that both algorithms approximate the best possible access strategies in cases of very low or very high client load (ClosestDly and Balanced, respectively). In addition, we showed that ShiftWtOpt and DelayBins converge quickly to ClosestDly and Balanced when these are the best access strategies, and in fact that DelayBins converges somewhat faster and more accurately. The reason for the superior performance of DelayBins in our experiments is best explained by the fact that decisions made by ShiftWtOpt are based on greater uncertainty than those made by DelayBins. This is due to the staleness of the response time information on which ShiftWtOpt's decisions are made.

The second differentiating factor between ShiftWtOpt and DelayBins is the client overhead. When the number of quorums (m) is small, both are efficient. However, when the number of quorums grows large, the client side overhead of ShiftWtOpt grows with the number of quorums, which can be exponential in the number of servers. In contrast, DelayBins enables a much more efficient implementation as m grows for the quorum systems we considered here, and thus is particularly advantageous when the number of servers is large.

We have experimented with several variations of ShiftWtOpt, but none seem to perform much better than the one described here. In general one faces the following trade-off when using this approach: either one makes the algorithm simple enough to be efficient (e.g., shifting weight between two quorums), but increases the convergence time by doing so. Or, one can be more aggressive, either when choosing the quorums or when choosing the number of pairs of quorums between which load is shifted, but in doing so the algorithm becomes more expensive.

Chapter 6

Conclusions

Edge computing represents one of the most successful architectures of our time for large scale Internet services. In this model, content is published at an origin server and then pushed to servers sitting at the edge of the network, close to the clients requesting it. This is particularly well-suited for static content and applications where the state does not change too frequently. At the same time, serving highly consistent dynamic content is still a topic of active research (e.g., Reiter and Samar [2007], Gao et al. [2005], Amza et al. [2003]).

One of the potential solutions to the problem of publishing highly consistent dynamic content in a scalable manner is to use quorum systems. Any quorum-based solution, however, needs to find a balance between several measures that affect performace on a wide-area network: network delay (latency), network congestion and server load. This thesis describes several methods for approximately minimizing the network measures when considered separately, and also methods for finding the right balance between network delay and server load so as to minimize client response time and maximize throughput.

In our treatment of these problems we look for the most general results. For instance in Chapter 2 our results refer to arbitrary quorum systems, arbitrary access strategies and arbitrary networks. The quorum selection algorithms in Chapters 4 and 5 are also designed for arbitrary quorum systems and networks, even though we evaluate them on specific topologies and for specific quorum constructions. In some cases we try to improve the general results for known quorum systems (in Chapter 2) and for specific networks (in Chapter 3).

There are several places where this thesis leaves room for further investigation. For instance, in Chapter 2 the problem of finding an optimal placement for any of the Crum-

bling Wall quorum systems (Peleg and Wool [1997]) is left open. In Chapter 3 we only briefly look at the benefits brought by migrating quorum elements between network nodes. We believe, however, that one can get substantial benefits in terms of congestion through the use of this technique (c.f., Westermann [2001]). Coupling migration with dynamically adapting the universe size (e.g., Alvisi et al. [2000], Kong et al. [2003]) may also enable quorum systems to better adapt to workload changes.

An interesting theoretical question left open in Chapter 5 is whether either of the ShiftWt and ShiftWtOpt quorum selection algorithms converges to Nash equilibrium in the case of a static workload. An answer in the affirmative to this question would also raise the issue as to how far is such a solution from the global optimum. Even more exciting (but probably harder) problems are those that consider these questions in a dynamic context.

Today's distributed services face an increased pressure from two contradictory goals: the need for better scalability or performance on one hand and the need for better reliability on the other. This thesis makes several steps towards finding a middle ground between these two goals. An important thing to note here however is that most of the work presented so far does not cosider failures. A very natural and interesting question that this thesis leaves open regards the extent to which the performance of the techniques developed here is affected by server or network failures.

Bibliography

M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005. 1.4, 4, 4.1, 4.1, 5, 5.1.2, 5.4

L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright. Dynamic Byzantine quorum systems. In *DSN '00: Proceedings of the International Conference on Dependable Systems and Networks*, pages 283–292, 2000. 6

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 *DSN '06: Proceedings of the 2006 International Conference on Dependable Systems and Networks*, pages 105–114, June 2006. 4.7, 5.4

Y. Amir and A. Wool. Evaluating quorum systems over the Internet. In *Proceedings of the 26th International Symposium on Fault-Tolerant Computing*, June 1996. 4.7, 5.4

Y. Amir and A. Wool. Optimal availability quorum systems: theory and practice. *Information Processing Letters*, 65(5):223–228, 1998. 2.5

C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Middleware*, pages 282–304, 2003. 6

O. Bakr and I. Keidar. Evaluating the running time of a communication round over the Internet. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing*, pages 243–252, July 2002. 4.7

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 *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, March 2004. 1.3, 4, 4.7, 5

R. A. Bazzi. Planar quorums. *Theoretical Computer Science*, 243(1-2):243–268, 2000. 2.5

R. A. Bazzi. Access cost for asynchronous Byzantine quorum systems. *Distributed Computing*, 14(1):41–48, 2001. 2.5

P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. Goldberg, Z. Hu, and R. Martin. Distributed selfish load balancing. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 354–363, 2006. 5.4

M. Bienkowski, M. Korzeniowski, and H. Räcke. A practical algorithm for cosntructing oblivious routing schemes. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 24–33, 2003. 3.2.1

P. Carmi, S. Dolev, S. Har-Peled, M. J. Katz, and M. Segal. Geographic quorum system approximations. *Algorithmica*, 41(4):233–244, 2005. 2.5

R. L. Carter and M. E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM '97: Proceedings of the 16th Annual Joint Conference of the IEEE Computer and Communications Societies*, page 1014, 1997. 5.4

M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), November 2002. 4.7

C. Chekuri and S. Khanna. On multi-dimensional packing problems. In *SODA '99: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 185–194, 1999. 3.5, 2

S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–592, 1992. 1.1, 1.4, 2.1.3, 2.1.1, 2.3, 2.3.1, 5, 5.1.2

M. Dahlin. Interpreting stale load information. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 285, 1999. 5.4

Y. Dinitz, N. Garg, and M. X. Goemans. On the single-source unsplittable flow problem. *Combinatorica*, 19(1):17–41, 1999. 1.2, 3.1.2, 3.2.2, 3.2.3, 3.7

S. Dolev, S. Gilbert, N. A. Lynch, A. A. Shvartsman, and J. L. Welch. Geoquorums: Implementing atomic memory in mobile *ad hoc* networks. In *DISC '03: Proceedings of the 17th International Symposium on Distributed Computing*, pages 306–320, 2003. 2.5

L. Engebretsen and J. Holmerin. Towards optimal lower bounds for clique and chromatic number. *Theoretical Computer Science*, 299(1-3):537–584, 2003. 3.5

A. W. Fu. Delay-optimal quorum consensus for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(1):59–69, 1997. 1.1, 2.5, 4, 4.2.1

L. Gao, M. Dahlin, J. Zheng, L. Alvisi, and A. Iyengar. Dual-quorum replication for edge services. In *Middleware 2005*, pages 184–204, 2005. 4, 6

D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 150–162, 1979. 1, 1.1, 2.1.3, 2.1.1, 2.3, 4.7, 5.1.2

S. Gilbert and G. Malewicz. The quorum deployment problem. In *8th International Conference on Principles of Distributed Systems (OPODIS'04)*, 2004. 2.5

D. Golovin, A. Gupta, B. Maggs, F. Oprea, and M.K. Reiter. Quorum placement in networks: Minimizing network congestion. In *Proceedings of the 25th ACM Symposium on Principles of Distributed Computing*, 2006. 4, 5.4

K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd Usenix/ACM SIGCOMM Internet Measurement Workshop (IMW)*, 2002. 1.3, 4.3, 5.1.1

A. Gupta, B. Maggs, F. Oprea, and M. K. Reiter. Quorum placement in networks to minimize delays. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, 2005. 4, 4.2.1, 5.4

J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated internet servers. *ACM SIGCOMM Computer Communication Review*, 25(4):288–298, 1995. 5.4

C. Harrelson, K. Hildrum, and S. Rao. A polynomial-time tree decomposition to minimize congestion. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 34–43, 2003. 3.2.1

M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990. 1

R. Holzman, Y. Marcus, and D. Peleg. Load balancing in quorum systems. *SIAM Journal on Discrete Mathematics*, 10(2):223–245, 1997. 5.4

J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: characterization and implications for cdns and web sites. In *WWW '02: Proceedings of the 11th International Conference on World Wide Web*, pages 293–304, 2002. 5

B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000. 2.2.3

N. Kobayashi, T. Tsuchiya, and T. Kikuno. Minimizing the mean delay of quorum-based mutual exclusion schemes. *Journal of Systems and Software*, 58(1):1–9, 2001. 1.1, 2.5, 4, 4.2.1

L. Kong, A. Subbiah, M. Ahamad, and D. M. Blough. A reconfigurable Byzantine quorum approach for the agile store. In *SRDS '03: Proceedings of the 22nd Symposium on Reliable Distributed Systems*, pages 219–, 2003. 6

A. Kumar, M. Rabinovich, and R. K. Sinha. A performance study of general grid structures for replicated data. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 178–185, 1993. 1.1, 1.4, 2.1.3, 2.1.1, 2.3, 2.3.1, 5, 5.1.2

L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2): 133–169, 1998. 4.7

T. Leighton, F. Makedon, and I. G. Tollis. A 2n-2 step algorithm for routing in an nxn array with constant size queues. In *SPAA '89: Proceedings of the 1st ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, 1989. 3.7

J. K. Lenstra, D. B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(3):259–271, 1990. 2.2.3

J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26(1):22–35, 1978. 2.2.2

X. Lin. Delay optimizations in quorum consensus. In *ISAAC '01: Proceedings of the 12th International Symposium on Algorithms and Computation*, pages 575–586, 2001. 1.1, 2.5, 4, 4.2.1, 4.4

B. M. Maggs, F. Meyer auf der Heide, B. Vocking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *IEEE Symposium on Foundations of Computer Science*, pages 284–293, 1997. 3.7

D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4): 203–213, 1998. 5.4

D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM Journal on Computing*, 29(6):1889–1906, 2000. 2.5, 5.4

D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001. 2.5

R. Mirchandaney, D. Towsley, and J. A. Stankovic. Analysis of the effects of delays on load sharing. *IEEE Transactions on Computers*, 38(11):1513–1525, 1989. 5.4

R. Mirchandaney, D. Towsley, and J. A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9(4):331–346, 1990. 5.4

M. Mitzenmacher. How useful is old information? (extended abstract). In *PODC '97: Proceedings of the 16th ACM Symposium on Principles of Distributed Computing*, pages 83–91, 1997. 5.4

M. Naor and A. Wool. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998. 2.1, 2.3.1, 5.4

T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM: Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002. 5.1.1

D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service placement in shared wide-area platforms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005. 4.7

F. Oprea and M. K. Reiter. Minimizing response time for quorum-system protocols over wide-area networks. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks*, June 2007. 5.4

V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 178–190, 2002. 5

D. Peleg and A. Wool. Crumbling walls: A class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997. 6

H. Racke. Minimizing congestion in general networks. In *IEEE Symposium on Foundations of Computer Science*, 2002. 1.2, 3.1.2, 3.2, 3.2.1, 3.7

P. Raghavan and C. D. Thompson. Provably good routing in graphs: regular arrays. In *STOC '85: Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 79–87, 1985. 3.7

S. Ranjan, R. Karrer, and E. W. Knightly. Wide area redirection of dynamic content by internet data centers. In *INFOCOM '04: Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004. 5.4

S. Ranjan, J. Rolia, E. Knightly, and H. Fu. Qos-driven server migration for internet data centers. In *IWQoS '02: Proceedings of the 10th International Workshop on Quality of Service*, 2002. 5

S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *INFOCOM '02: Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002. 5.4

M. K. Reiter and A. Samar. Quiver: Consistent and scalable object sharing for edge services. *IEEE Transactions on Parallel and Distributed Systems*, 2007. 6

N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992. 5.4

D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(3):461–474, 1993. 1.1, 2.2.3, 2.2.12

S. Sivasubramanian, G. Pierre, and M. van Steen. Autonomic data placement strategies for update-intensive web applications. In *AAA-IDEA '05: Proceedings of the First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications*, pages 2–9, 2005. 1

A. Srinivasan. Distributions on level-sets with applications to approximation algorithms. In *FOCS '01: Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science*, page 588, 2001. 1.2, 3.1.2, 3.5.1, 3.5.1

D. Starobinski and T. Wu. Performance of server selection algorithms for content replication networks. In *NETWORKING*, pages 443–454, 2005. 5.4

R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979. 1, 1.1, 2.1.3, 2.1.1, 2.3, 4.7, 5.1.2

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*, 10(4):337–345, 1999. 1.1, 2.5, 4

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 Operating Systems Review*, 36(SI):271–284, 2002. 4, 4.1, 5, 5.1.2

L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982. 3.7

L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *STOC '81: Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 263–277, 1981. 3.7

V. V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, 2001. 2.2.3

M. Westermann. *Caching in Networks: Non-Uniform Algorithms and Memory Capacity Constraints*. Dissertation, Universität Paderborn, Heinz Nixdorf Institut, Theoretische Informatik, 2001. 3.6.1, 3.7, 6

G. J. Woeginger. On the approximability of average completion time scheduling under precedence constraints. In *Proceedins of the 28th International Colloquium on Automata, Languages and Programming (ICALP' 2001)*, LNCS 2076, pages 862–874, 2001. 2.2.2, 2.2.5

C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *ATEC '97: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 8–8, 1997. 5.4

H. Yu. Signed quorum systems. In *Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing*, pages 246–255, 2004. 2.5