Network delay-aware load balancing in selfish and cooperative distributed systems

Piotr Skowron
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
Email: p.skowron@mimuw.edu.pl

Krzysztof Rzadca
Faculty of Mathematics, Informatics and Mechanics
University of Warsaw
Email: krzadca@mimuw.edu.pl

***Abstract*—We consider a geographically distributed request processing system composed of various organizations and their servers connected by the Internet. The latency a user observes is a sum of communication delays and the time needed to handle the request on a server. The handling time depends on the server congestion, i.e. the total number of requests a server must handle. We analyze the problem of balancing the load in a network of servers in order to minimize the total observed latency. We consider both cooperative and selfish organizations (each organization aiming to minimize the latency of the locally-produced requests). The problem can be generalized to the task scheduling in a distributed cloud; or to content delivery in an organizationally-distributed CDNs.**

In a cooperative network, we show that the problem is polynomially solvable. We also present a distributed algorithm iteratively balancing the load. We show how to estimate the distance between the current solution and the optimum based on the amount of load exchanged by the algorithm. During the experimental evaluation, we show that the distributed algorithm is efficient, therefore it can be used in networks with dynamically changing loads.

In a network of selfish organizations, we prove that the price of anarchy (the worst-case loss of performance due to selfishness) is low when the network is homogeneous and the servers are loaded (the request handling time is high compared to the communication delay). After relaxing these assumptions, we assess the loss of performance caused by the selfishness experimentally, showing that it remains low.

Our results indicate that a set of servers handling requests, connected in a heterogeneous network, can be efficiently managed by a distributed algorithm. Additionally, even if the network is organizationally distributed, with individual organizations optimizing performance of their requests, the network remains efficient.

I. Introduction

One of the most important aspects affecting the perceived quality of a web service is the delay in accessing the content. To avoid servers' congestion, the content of the web pages is commonly replicated in multiple locations. Additionally, in order to minimize the network latency, the replicas are placed close to users. Because the intensity of the web traffic changes dynamically, efficient mirroring requires both expensive infrastructure and effective load balancing algorithms. As the result many organizations decide to handle the task of mirroring their data to dedicated platforms — content delivery networks (CDNs) [18], [29]. The CDNs have been very successful in the recent years: Akamai [25], [27], [34], the largest CDN, handles around 15-20% of the Internet traffic.

Consider an apparently different distributed system: a cloud of datacenters performing computationally-intensive parallel calculations. Each datacenter attempts to accelerate its calculations by distributing some of its load to less loaded and faster datacenters. However, the datacenters in remote locations must be avoided as the time needed to transfer the input and the result may dominate the processing time.

Routing the requests in a CDN and distributing the load in a cloud are strongly related problems. In both there are systems of servers connected by the network (for simplicity, in cloud, we refer to a single datacenter as a "server", as we will not explore the parallelism inside a datacenter). The handling time on a server depends both on its performance metrics and its load. The final perceived latency comes from the network delays (required for transmitting the input data and the result) and from the handling time on the servers. Finally, in both cases every server has its initial load: in a CDN, the load is the current number of the data access requests to the server; in a cloud, the load is the number of initial tasks. We generalize these two problems to a load balancing of remote services.

We assume that in the balanced system, the handling time of a single request on a server linearly depends on the total number of requests to be processed by the server. A linear dependency reflects a constant throughput of a server. In real systems, increasing the level of concurrency too much may overload the server decreasing its throughput (trashing). However, assuming that the amount of work in the system as a whole is reasonable, there should be no overloaded servers in the balanced state. Similar assumptions are usually taken in congestion games [14], [31] and in the queuing theory, where a linear dependency is expressed by Little's law.

We assume that the transmission duration of a single request does not depend on the number of sent requests. Although some models (e.g., routing games [26]) consider the cases when the bandwidth of a link may become a bottleneck, we focus on a widespread network, in which there are multiple cost-comparable routing paths between the servers. Thus, sending any data from one server to another should not significantly increase the network delay between them. These assumptions are also justified by our experiments – in

Appendix we discuss how the intensity of the network load, generated between the servers, influences the RTT between the servers in PlanetLab environment. In other words, we consider that the load our system imposes on the network is negligible: thus, the network delay is caused only by the latency (resulting from e.g., geographical distribution). Our problem formulation assumes the knowledge of such latencies; this is not a limitation because monitoring the pairwise latencies, which can change in time, is a well studied problem with known solutions (e.g. see [11], [35]). Optimizing latency is important for instance when streaming video files: a large latency delays the start, and, in case of communication problems, can be perceived as breaks in transmission.

Balancing of the servers loads and finding the mirroring minimizing network delays are analyzed in the literature, but usually separately (see Section VIII). Distributed systems should consider both communication and computation. On one hand, clouds get geographically distributed, thus cannot ignore network latency. On the other, a CDN handling complex, dynamically-created content of the modern web, can no longer ignore the load imposed by the requests.

For delivering large static content, like multimedia, some currently used techniques cache the content at specially designated front-end servers [12]. In this case, our algorithms can be viewed as a complementary optimization technique to caching – once the content must be downloaded from the back-end servers, we show how to efficiently distribute the download requests.

In cloud computing, our model fits for instance processing streams of data in the real time or when the data stream is continuously produced and too large to be processed off-line. Consider a user interacting with a simulated virtual environment (e.g. [4]): user's actions are captured by cameras; their image streams are analyzed in the real time to build a 3D model; then this model interacts with the virtual world model. Other applications include extracting statistics on users' actions in the Internet; or image analysis.

In addition to a classic system with a central management, we analyze an organizationally-distributed system. Instead of a single, system-wide goal (minimize the overall request handling time), the organizations are selfishly interested only in optimizing the handling time of their local requests. This model reflects a CDN created as an agreement between e.g., many ISPs; or a federation of clouds, each having a different owner. Because typically the load changes dynamically, with peaks of demand followed by long periods of low activity, individual organizations are motivated to enter such a system: a peak can be offloaded, whereas handling foreign requests in the period of low activity is relatively inexpensive.

The lack of central coordination in the organizationally-distributed system increases the average processing time. The price of anarchy [23] expresses the worst-case relative increase in the latency in comparison with relinquishing the control to a centrally-managed organization (like Akamai's CDN). As the price of anarchy varies considerably between systems (from relatively small in congestion games to unbounded in selfish replication [32]), we were curious to check it in our system.

Our contribution is the following: (i) We show that the problem of network delay-aware load balancing can be stated as an optimization problem in the continuous domain; we prove that the optimization problem is convex and, in particular, polynomially solvable. This result also indicates that the local optimization techniques can be applied to the problem. (ii) We propose a distributed algorithm that iteratively balances the servers' load towards the optimum. We confirm the algorithm's efficiency through theoretical analysis and in simulations. (iii) We show how the problem can be extended and, in particular, how it can be applied to find the optimal distribution of the data (even with their replicas) when the access pattern of the users is known (Section VII). (iv) In a network of selfish servers, after some simplifying assumptions, we prove that the price of anarchy is low ($1+O(2cs/l_{av})$). The experiments show that the loss of performance caused by the selfishness remains low (below 1.15) also without these assumptions.

II. Mathematical model

Organizations, servers, tasks The system consists of a set of *m organizations*, each owning a *server* (or a *cluster*) connected to the Internet. The servers are uniform; each server i has a constant processing speed s_i. The i-th organization has its *own load* consisting of a large number n_i of small, individual *tasks* (or *requests*). The amount of load n_i can be considered as a number of tasks at a particular time moment (snapshot); or, alternatively, as a steady state rate of incoming requests in a system continuously processing requests. A task corresponds to, e.g., in a computational cloud, a unit-size computation (e.g.: a single work unit in a BOINC-type application; or a single invocation of a map-reduce function); or, in a CDN, a request for remote data coming from a user assigned to server i (typically, a user would be assigned to the closest server). In the basic model we assume that the small tasks have the same sizes (e.g. this corresponds to the divisible computation load; or in a CDN to the case where the stored data chunks have constant sizes); thus the execution of the single request on the i-th server takes $1/s_i$ time units. In Section VII we show how to easily extend our results to the tasks of different sizes.

Relaying tasks, communication delays Each organization can relay some of its own requests to other servers. If the request is relayed, the observed handling time is increased by the communication latency on the link. We denote the communication latency between i-th and j-th server as c_{ij} (with $c_{ii}=0$). Since communication delay of a single request does not depend on the amount of exchanged load (which

is explained in Section I and which is confirmed by our experiments on PlanetLab – see Appendix) c_{ij} is just a constant instead of a function of the network load. We assume that the routing in the system is correct (optimized by the network layer). Thus, we will not consider optimizing communication time by relaying requests from i to j through a third server k (if $c_{ik}+c_{kj}<c_{ij}$, the network layer would also discover the route $i \to k \to j$ and update the routing table accordingly, so that $c_{ij} := c_{ik}+c_{kj}$). We assume that each request can be sent to and executed on any server. However, if we set some of the communication delays to infinity, we restrict the basic model to the case when each organization is allowed to relay its requests only to the given subset of the servers (its neighbors) which models e.g. the trust relationship.

Relay fractions, current loads We use a fractional model in which a *relay fraction* ρ_{ij} denotes the fraction of the i-th organization's own requests that are sent (relayed) to be executed at j-th server ($\forall_{i,j}\,\rho_{ij} \geq 0$ and $\forall_i \sum_{j=1}^{j=m} \rho_{ij} = 1$). The load balancing problem is to find the appropriate values of the relay fractions (formalized in the further part of this Section). Once the fractions are known, each organization knows how many of its own requests it should send to each server; the tasks are sent and executed at the appropriate servers. The fractional model might be considered as a relaxation of a problem of handling non-divisible requests; in Section VII we show how to round the solution of a fractional model to a discrete model. Moreover, the fractional model itself fits the divisible load model used in the scheduling theory. For the sake of the clarity of the presentation we use the additional notation for the number of requests redirected from server i to j – r_{ij} (thus $r_{ij} = n_i\rho_{ij}$), and for the *current load* of the server i, i.e. the number of requests relayed to i by all other organizations, including the organization owning the server itself – l_i (thus, $l_i = \sum_{j=1}^{m} r_{ji}$).

Completion times We don't assume any particular order of requests executed on a server. First, since the number of requests is large, considering any particular order on the servers would increase the computational complexity. Second, in a continuously running systems, we have no control over the order in which requests are produced (especially as they can be also delayed by the network); the usual FIFO policy results in an arbitrary order. Thus, for each of the l_j request that are actually processed on j-th server, the expected processing time of each request is equal to $1/l_j \sum_1^{l_j} i/s_j = l_j/2s_j$ (constant omitted for clarity). Since i-th organization relayed r_{ij} requests to j, the expected total completion time of requests relayed by i to j is equal to $r_{ij}(l_j/2s_j + c_{ij})$. The expected total processing time C_i of the i-th organization's own requests is a sum over all servers j of the expected total completion times of the requests owned by i and relayed to j.

$$C_i = \sum_{j=1}^{m} \left(\frac{l_j}{2s_j} + c_{ij} \right) r_{ij} = \sum_{j=1}^{m} \left(\left(c_{ij} + \sum_{k=1}^{m} \frac{\rho_{kj} n_k}{2s_j} \right) \rho_{ij} n_i \right). \tag{1}$$

We consider the expected (or the average) processing time, rather than the makespan of an organization for several reasons. The average processing time is similar to the widely-used sum of processing times criterion (ΣC_i). We assume that the workload of each organization is created by many users. ΣC_i models users' performance better than the makespan [17]. In all the contexts motivating our work from Section I (e.g., processing streams of data in the real time, delivering content to the users) we are focused on the average user performance. Also, while C_i depends on the vector $\rho = [\rho_{kl}]$ quadratically, the relation between the makespan and ρ is just linear, which makes the problem considerably easier. Thus, we believe that some of our results could be adapted for the cases when some different from the pointed applications of our model would require optimizing the makespan.

The total processing time of all the requests in the system is denoted as $\sum C_i = \sum_{i=1}^{i=m} C_i$.

Problem formulation We consider two related problems. First, the goal is to find such a vector of the fractions, ρ, that the total processing time of the requests, $\sum C_i$, is minimized. This goal corresponds to a centrally-managed system having a unique owner and a single goal.

Second, we analyze the case when servers are a common good, but each organization is selfishly minimizing the processing time of its own requests. The i-th organization is responsible for sending its own requests to appropriate servers. In other words, the i-th organization adjusts the values of ρ_{ij} in order to minimize C_i. This approach is similar to the selfish job model [37], in which jobs selfishly choose processors to minimize their execution time. Similar agreements exist in real-life systems: e.g., PlanetLab servers are treated as a common good managed by a central entity; PlanetLab users choose the servers they want to use for their experiments. Also, in academic grids (e.g. Grid5000 in France), participating organizations grant control over their resources to a central entity; in return, users can submit their jobs to any resource. In this case, we look for such a vector of the fractions ρ for which the system reaches the Nash equilibrium. By comparing the resulting $\sum C_i$ with the result for the centrally-managed system, we will find the price of anarchy, quantifying the effect of selfishness on the total processing time.

III. Optimal solution

In this section we assume that there is a central processing unit that has the complete knowledge about the whole system. Given the communication latencies c_{ij} and the organizations' own loads n_i, our goal is to find an algorithm

setting relay fractions ρ_{ij} so that the total processing time of all the requests $\sum C_i$ is minimized. Below we show that the problem can be stated as a quadratic programming problem with a positive-definite Q matrix. This means that the optimization problem is convex and, in particular, it is polynomially solvable. We consider this result powerful, as it indicates that any local optimization techniques can be applied to the problem. In particular, in the next section we present a distributed algorithm that, in practice, is very efficient.

Theorem 1. *The problem of minimizing $\sum C_i$ can be expressed as the quadratic programming problem: $\sum C_i = \rho^T Q \rho + b^T \rho$, with the positive-definite Q matrix. In particular, this means that the minimization problem is convex.*

Proof: We express the total processing time $\sum C_i$ in a matrix form as $\sum C_i = \rho^T Q \rho + b^T \rho$, where:

- ρ is a vector of *relay* fractions with $m \cdot m$ elements. $\rho_{(i,j)}$, the element at $(i \cdot m + j)$-th position, denotes the fraction of local requests of i-th server that are relayed to j-th server ρ_{ij}, thus:
  $\rho = [\rho_{(1,1)}, \rho_{(1,2)}, \ldots, \rho_{(1,m)}, \rho_{(2,1)}, \ldots, \rho_{(m,m)}]^T$;
- Q is m^2-by-m^2 matrix in which $q_{(i,j),(k,l)}$ denotes the element in $(i \cdot m + j)$-th row and in $(k \cdot m + l)$-th column:

$$q_{(i,j),(k,l)} = \begin{cases} n_i n_k / s_j & \text{if } j = l \text{ and } i < k; \\ n_i n_k / 2 s_j & \text{if } j = l \text{ and } i = k; \\ 0 & \text{otherwise;} \end{cases} \quad (2)$$

  Figure 1 presents the structure of matrix Q.
- b is a vector with m^2 elements with b_{ij} denoting an element at $(i \cdot m + j)$-th position: $b_{(i,j)} = c_{ij} n_i$.

The following derivation shows how the matrix Q is constructed:

$$\rho^T Q \rho = \sum_{i,j} \rho_{(i,j)} \sum_{k \geq i} q_{(i,j),(k,j)} \rho_{(k,j)} \quad (3)$$

$$= \sum_{i,j} \rho_{(i,j)} \left(\sum_{k>i} \frac{n_i n_k \rho_{(k,j)}}{s_j} + \frac{n_i^2 \rho_{(i,j)}}{2 s_j} \right) \quad (4)$$

$$= \sum_i \sum_j \sum_k \frac{n_i n_k \rho_{(i,j)} \rho_{(k,j)}}{2 s_j} = \sum_i \sum_j \frac{r_{ij} l_j}{2 s_j}. \quad (5)$$

(3) follows from the construction of the matrix Q (only elements $k \geq i$ are non-zero). (4) substitutes $q_{(i,j),(k,l)}$ with the values defined in (2). (5) uses commutativity of multiplication and substitutes $l_j = \sum_k n_k \rho_{(k,j)}$ and $r_{ij} = n_i \rho_{(i,j)}$.

The constraints that ρ_{ij} are the fractions ($\forall_{i,j} \rho_{ij} \geq 0$ and $\forall_i \sum_{j=1}^{j=m} \rho_{ij} = 1$) can also be expressed in the matrix form. First, $\rho \geq 0_{m^2}$, where 0_{m^2} is a vector of length m^2 consisting of zeros. Second, $A\rho = 1_m$, where 1_m is a vector of length m and consisting of ones, and A is a m-by-m^2 matrix defined by the following equation:

$$a_{ij} = \begin{cases} 1 & \text{if } im \leq j < (i+1)m \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

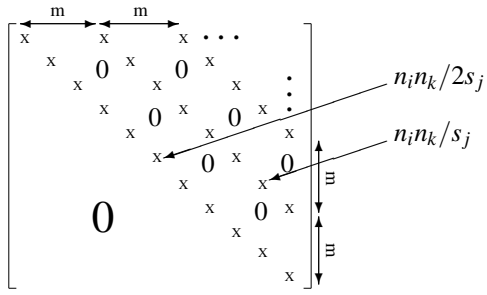

Figure 1: Matrix Q: X denotes non-zero values

Algorithm 1: CALCBESTTRANSFER(i, j)

input: (i, j) – the identifiers of the two servers
Data: $\forall_k\ r_{ki}$ – initialized to the number of requests owned by k and relayed to i ($\forall_k\ r_{kj}$ is defined analogously)
Result: The new values of r_{ki} and r_{kj}
foreach k **do**
  $r_{ki} \leftarrow r_{ki} + r_{kj}$; $r_{kj} \leftarrow 0$;
end
$l_i \leftarrow \sum_k r_{ki}$; $l_j \leftarrow 0$;
servers $\leftarrow$ sort $[k]$ so that $c_{kj} - c_{ki} < c_{k'j} - c_{k'i} \implies k$ is before k';
foreach $k \in$ *servers* **do**
  $\Delta r_{ikj} \leftarrow \min\left(\frac{(s_j l_i - s_i l_j) - s_i s_j (c_{kj} - c_{ki})}{(s_i + s_j)}, r_{ki} \right)$;
  if $\Delta r_{ikj} > 0$ **then**
    $r_{ki} \leftarrow r_{ki} - \Delta r_{ikj}$; $r_{kj} \leftarrow r_{kj} + \Delta r_{ikj}$;
    $l_i \leftarrow l_i - \Delta r_{ikj}$; $l_j \leftarrow l_j + \Delta r_{ikj}$;
  end
end
return *for each k: r_{ki} and r_{kj}*

Algorithm 2: Min-Error (MinE) algorithm performed by server id.

Notation: `impr` (i, j) $\leftarrow$ this function calculates the improvement of $\sum C_i$ when transferring requests between i and j. The number of requests that should be transferred can be computed by `calcBestTransfer`(i, j).
partner $\leftarrow$ argmax$_j$(`impr`(id, j));
`rely` (id, partner, `calcBestTransfer`(*id, partner*));

Minimization of $\sum C_i(\rho) = \rho^T Q \rho + b^T \rho$ with constraints $\rho \geq 0_{m^2}$ and $A\rho = 1_m$ is an instance of quadratic programing problem. As an upper triangular matrix, matrix Q has m^2 eigenvalues equal to the values at the diagonal: $n_i^2 / 2 s_j$ ($1 \leq i, j \leq m$). All eigenvalues are positive so Q is positive-definite. ■

Corollary 1. *The problem of minimizing $\sum C_i$ is polynomially solvable.*

According to [22], the best running time reported for solving quadratic programing problem with linear constraints is $O(n^3 L)$ [21], where L represents the total length of the input coefficients and n the number of variables (here $n = m^2$), so the complexity of the best solution is $O(Lm^6)$. However, Theorem 1 encourages to apply the local optimization techniques, like the algorithm presented in the next section.

IV. Distributed algorithm

The centralized algorithm requires the information about the whole network – the size of the input data is $O(m^2)$ and the Q matrix has $O(m^3)$ non-zero entries. A centralized algorithm has thus the following drawbacks: (i) collecting information about the whole network is time-consuming; moreover, loads and latencies may frequently change; (ii) a standard solver takes significant time (recall $O(Lm^6)$ in Section III); (iii) the central algorithm is more vulnerable to failures. Motivated by these limitations we introduce a distributed algorithm for finding the optimal solution.

The distributed algorithm requires that each server has up-to-date information about the loads on the other servers and about the communication delays from itself to the other servers (and not for all pairs of servers). Thus, for each server, the size of the input data is $O(m)$. As indicated in Section I, the problem of monitoring the latencies is well-studied. The loads can be disseminated by a gossiping algorithm. As gossiping algorithms have logarithmic convergence time, if the gossiping is executed about $O(log(m))$ times more frequently than our algorithm, each server has accurate information about the loads.

Each organization, i, keeps for each server, k, the information about the number of requests that were relied to i by k. The algorithm iteratively improves the solution – the i-th server in each step communicates with the locally optimal partner server – j (Algorithm 2). The pair (i, j) locally optimizes the current solution by adjusting, for each k, r_{ki} and r_{kj} (Algorithm 1). In the first loop of the Algorithm 1, i, one of the servers, takes all the requests that were previously assigned to i and to j. Next, all the organizations $[k]$ are sorted according to the ascending order of $(c_{kj} - c_{ki})$. The lower the value of $(c_{kj} - c_{ki})$, the more profitable it is to run requests of k on j rather than on i in terms of the network topology. Then, for each k, the loads are balanced between servers i and j.

In Section III we have shown that the optimization problem is convex. Thus, it is natural to try local optimization techniques. The presented mechanism requires only two servers involved in each optimization step, thus it is very robust to failures. This mechanism is similar in spirit to the diffusive load balancing [1], [2], [7]; however there are substantial differences related to the fact that the machines are geographically distributed: (i) In each step no real requests are transferred between the servers; this process can be viewed as a simulation run to calculate the relay fractions ρ_{ij}. Once the fractions are calculated the requests are transfered and executed at the appropriate server. (ii) Each pair (i, j) of servers exchanges not only its own requests but the requests of all servers that relayed their requests either to i or to j. Since different servers may have different communication delays to i and j the local balancing requires more care (Algorithms 1 and 2).

Algorithm 2 has the following properties: (i) The single optimization step requires only two servers to be available (thus, it is very robust to failures). (ii) Any algorithm that in a single step involves only two servers cannot perform better (Theorem 2). (iii) The algorithm does not require any requests to be unnecessarily delegated – once the relay fractions are calculated the requests are sent over the network; also, each request is transferred over the network at most once. (iv) The complexity of the algorithm does not depend on the number of requests (only on the number of servers), thus it is suitable for the systems with a large number of small requests. (v) In each step of the algorithm we are able to estimate the distance between the current solution and the optimal one (Proposition 1).

A. Correctness

The following Lemma shows how to optimally exchange the requests owned by organization k between a pair of servers i and j.

Lemma 1. *Consider two servers i and j that execute r_{ki} and r_{kj} requests of the k-th organization. The total processing time, $\sum C_i$, is minimized when the k-th server relies Δr_{ikj} from r_{ki} requests to be additionally executed on j-th server:*

$$\Delta r'_{ikj} = \frac{(s_j l_i - s_i l_j) - s_i s_j (c_{kj} - c_{ki})}{(s_i + s_j)}$$

$$\Delta r_{ikj} = \max(0, \min(r_{ki}, \Delta r'_{ikj}))$$

Proof: If the k-th server moves some of its requests from i to j, then it affects the completion time of all requests that were relayed either to i or to j (initial requests of all servers). Recall that l_i and l_j are the loads of the servers, respectively, i and j, that is they include all tasks relayed to, respectively, i and j. Thus, if k removes Δr of its requests from i, then the new processing time of all tasks on the server i will be $(l_i - \Delta r)^2/2s_i$. Thus, we want to find Δr_{ikj} that minimizes the function f:

$$f(\Delta r) = \frac{(l_i - \Delta r)^2}{2s_i} + \frac{(l_j + \Delta r)^2}{2s_j} - \Delta r c_{ki} + \Delta r c_{kj}$$

We can find minimum by calculating derivative:

$$\frac{df}{d\Delta r} = \frac{\Delta r - l_i}{s_i} + \frac{\Delta r + l_j}{s_j} - c_{ki} + c_{kj} = 0$$

$$\Delta r_{ikj} = \frac{(s_j l_i - s_i l_j) - s_i s_j (c_{kj} - c_{ki})}{(s_i + s_j)}$$

Also $\Delta r \in \langle 0, r_{ki} \rangle$, which proves the thesis. ∎

The following lemma proves the correctness of Algorithm 1.

Theorem 2. *After execution of Algorithm 1 for the pair of servers i and j, it is not possible to improve $\sum C_i$ only by exchanging any requests between i and j.*

Sketch of Proof: First we show that after the second loop no requests should be transferred from i to j. For each organization k the requests owned by k were transferred from i to j in some iteration of the second loop; also, each of the next iterations of the second loop could only cause the increase of the load of j (and decrease of i); thus transferring more requests of k from i to j would be inefficient. Second, we will show that after the second loop no requests should be transferred back from j to i either. Let us take the last iteration of the second loop in which the requests of some organization k were transferred from i to j. After this transfer we know that $\Delta r_{ikj} = \frac{(s_j l_i - s_i l_j) - s_i s_j (c_{kj} - c_{ki})}{(s_i + s_j)} \geq 0$ (otherwise the transfer would not be optimal). However, this implies that $\Delta r_{ik'j} = \frac{(s_j l_i - s_i l_j) - s_i s_j (c_{k'j} - c_{k'i})}{(s_i + s_j)} \geq 0$ for each server k' considered before k. As $\Delta r_{ik'j} \geq 0$ we get $\Delta r_{jk'i} \leq 0$. □

B. Error estimation

The following analysis bounds the distance of the current solution of the distributed algorithm to the optimum as a function of the disparity of servers' load. When running the algorithm, this result can be used to assess whether it is still profitable to continue running the algorithm: if the load disparity is low, the current solution is close to the optimum.

We introduce the following notation for the analysis. ρ' is the snapshot (the current solution) derived by distributed algorithm. ρ is the optimal solution that minimizes $\sum C_i$ (if there are multiple optimal solutions with the same $\sum C_i$, ρ is the closest solution to ρ' in the Manhattan metric). $(P, \Delta\rho)$ is a weighted, directed *error graph*: $\Delta\rho[i][j]$ indicates the number of requests that should be transferred from server i to j in order to reach ρ from ρ' ($\Delta\rho[i][j]$ requests either belong to i, or to j, and not to another server k). We define dir as the *direction of transport*: $\mathrm{dir}(i,j) = 1$ if i transfers to j its own requests; $\mathrm{dir}(i,j) = -1$ if i returns to j the requests that initially belonged to j. Let $succ(i)$ denotes the set of successors in the error graph: $succ(i) = \{j : \Delta\rho[i][j] > 0\}$; $prec(i)$ denotes the set of predecessors: $prec(i) = \{j : \Delta\rho[j][i] > 0\}$.

In the error graph, a *negative cycle* is a sequence of servers $i_1, i_2, \ldots, i_n$ such that (i) $i_1 = i_n$; (ii) $\forall_{j \in \{1,\ldots n-1\}}$ $\Delta\rho[i_j][i_{j+1}] > 0$; and (iii) $\sum_{j=1}^{n-1} \mathrm{dir}(i_j, i_{j+1}) c_{i_j i_{j+1}} < 0$.

A negative cycle is sequence of servers that essentially redirect their requests to one another. A solution without negative cycles has a smaller processing time: after dismantling a negative cycle, loads on servers remain the same, but the communication time is reduced. In Appendix, we show how to detect and remove negative cycles; in order to simplify the presentation of the subsequent analysis, we consider that there are no negative cycles.

Proposition 1. *If (i) the error graph $\Delta\rho$ has no negative cycle; and (ii) $\sum_j \max_k((\frac{1}{s_j} + \frac{1}{s_k})\Delta r_{jk}) = \Delta R$ (Δr_{ij} is the number of requests which in the current state ρ' would be relied to j-th server by the i-th server (as the result of Algorithm 1), then $\|\rho - \rho'\|_1 \leq (4m+1)\Delta R \sum_i s_i$, where $\|\cdot\|_1$ denotes the Manhattan metric.*

Proof: The proof is presented in the full version of the article [33]. ■

Proposition 1 gives the estimation of the error for such partial solutions that do not have negative cycles. Therefore the algorithm that cancels negative cycles (see Appendix) should be run whenever the estimation for distance to the optimal solution is needed. Our experiments show, however, that the negative cycles are rare in practice and that pure Algorithm 2 can remove them efficiently (Section VI).

V. Selfish organizations

In this section we consider the case when the organizations are acting selfishly – the i-th of them tries to minimize the total processing time of its own requests – C_i. We are interested in a steady state in which all the peers have no interest in redirecting any of its requests to different servers – the Nash equilibrium.

A. Homogeneous network

In this section we present the characteristic of the Nash equilibrium in case when all the servers have equal processing power ($\forall_i\ s_i = s$), and when all the connections between servers have the same communication delay ($\forall_{ij}\ c_{ij} = c$). We consider homogeneous model, as the modeling of a heterogeneous interconnection graph is complex. The simulation experiments (Section VI-C) show that in the case of selfish servers the average relative degradation of the system goal on heterogeneous networks is similar to, or lower than on the homogeneous networks.

Lemma 2. *For every two servers i and j the difference between their average loads is bounded: $|l_i - l_j| \leq c \cdot s$*

Proof: (by contradiction) Assume $|l_i - l_j| > c \cdot s$. Without loosing the generality, $l_i > l_j$. Recall that r_{ij} is the number of redirected requests $r_{ij} = n_i \rho_{ij}$. For each sever k ($k \neq i$), it is not profitable to put more of its requests to the more loaded server, so $r_{kj} \geq r_{ki}$. Now we want to find the relation between l_i, l_j, r_{ij} and r_{ii}. In a Nash equilibrium, it is not profitable for i to redirect any additional x of its own requests from itself to j, which can be formally expressed by the equation:

$$0 \leq \frac{(l_i - x)(r_{ii} - x)}{2s} + \frac{(l_j + x)(r_{ij} + x)}{2s} + c(r_{ij} + x) - \frac{l_i r_{ii}}{2s} - \frac{l_j r_{ij}}{2s} - c r_{ij},$$

equivalent to:

$$r_{ij} - r_{ii} + 2x \geq l_i - l_j - 2c \cdot s.$$

Because the inequality must hold for every positive x, and because $l_i - l_j > c \cdot s$

$$r_{ij} - r_{ii} > c \cdot s - 2c \cdot s = -c \cdot s$$

Now we can show the contradiction, because

$$l_j = \sum_{k=1}^{k=m} r_{kj} > \sum_{k=1}^{k=m} r_{ki} - c \cdot s = l_i - c \cdot s$$

from which it follows that

$$l_i - l_j < c \cdot s.$$

■

Let us denote the average load on the server as l_{av}, thus $l_{av} = \frac{1}{m}\sum_{i=1}^{i=m} l_i$. The following theorem gives the tight estimation of the price of anarchy when the servers are loaded compared to the delay ($l_{av} \gg 2cs$). If the servers are not loaded then the thesis of Theorem 3 is still correct, but our estimation of the price of anarchy is dominated by $O((\frac{c}{l_{av}})^2)$ element, thus it is not tight).

Theorem 3. *The price of anarchy in the homogeneous network is:* $PoA = 1 + \frac{2cs}{l_{av}} + O((\frac{cs}{l_{av}})^2)$.

Proof: (upper bound) We denote the load imbalance on the i-th server as $\Delta_i = l_i - l_{av}$. It follows that $\sum_{i=1}^{i=m} \Delta_i = 0$. Also, from Lemma 2 we have $\Delta_i \leq c \cdot s$. Additionally, each request can be relied at most once, thus the total time used for communication is bounded by $ml_{av}c$. Therefore, the total processing time in case of selfish peers, $\sum C_i(\text{self})$ is bounded:

$$\sum C_i(\text{self}) \leq ml_{av}c + \sum_i \frac{(l_i)^2}{2s} = ml_{av}c + \sum_i \frac{(l_{av}+\Delta_i)^2}{2s}$$
$$= \frac{ml_{av}^2}{2s} + \sum_i \frac{\Delta_i^2}{2s} + ml_{av}c \leq \frac{ml_{av}^2}{2s} + \frac{mc^2 s}{2} + ml_{av}c$$

The total processing time is the smallest when the servers have equal load (each server processes exactly l_{av} requests) and do not communicate, thus the optimum is bounded by $(\sum C_i)^* \geq \frac{ml_{av}^2}{2s}$.

Thus, the price of anarchy is bounded by:

$$PoA \leq \frac{ml_{av}^2 + 2ml_{av}cs + mc^2s^2}{ml_{av}^2} = 1 + \frac{2cs}{l_{av}} + (\frac{cs}{l_{av}})^2$$

(tightness) Consider an instance with servers having equal initial load: $\forall_i\, n_i = l_{av}$.

In the optimal solution no requests will be redirected.

When servers are selfish, the i-th server will redirect to j-th server ($i \neq j$) $\frac{l_{av}-2c\cdot s}{m}$ requests and will execute $(2c\cdot s + \frac{l_{av}-2c\cdot s}{m}$ of its own requests on itself. As a result: $l_i = l_{av}$.

This is a Nash Equilibrium state, because it is not profitable for any server to redirect any x more of its own requests to the other server, nor to execute any x more requests on itself instead of some other server, as the two following inequalities hold for every positive x:

$$0 < \frac{l_{av}-x}{2s}(2c\cdot s + \frac{l_{av}-2c\cdot s}{m} - x) + \frac{l_{av}+x}{2s}(\frac{l_{av}-2c\cdot s}{m} + x)$$
$$+ cx - \frac{l_{av}}{2s}(2c\cdot s + \frac{l_{av}-2c\cdot s}{m}) - \frac{l_{av}}{2s}(\frac{l_{av}-2c\cdot s}{m})$$
$$0 < \frac{l_{av}+x}{2s}(2c\cdot s + \frac{l_{av}-2c\cdot s}{m} + x) + \frac{l_{av}-x}{2s}(\frac{l_{av}-2c\cdot s}{m} - x)$$
$$- cx - \frac{l_{av}}{2s}(2c\cdot s + \frac{l_{av}-2c\cdot s}{m}) - \frac{l_{av}}{2s}(\frac{l_{av}-2c\cdot s}{m})$$

Thus, we get the lower bound on the price of anarchy:

$$PoA \geq \frac{ml_{av}^2 + m(l_{av} - 2c\cdot s - \frac{l_{av}-2c\cdot s}{m})c\cdot 2s}{ml_{av}^2}$$
$$= 1 + \frac{2cs}{l_{av}} - 4(\frac{sc}{l_{av}})^2 - \frac{2(l_{av}-2c^2s^2)}{ml_{av}^2} \geq 1 + \frac{2cs}{l_{av}} - 4(\frac{cs}{l_{av}})^2.$$

Summarizing:

$$1 + \frac{2cs}{l_{av}} - 4(\frac{cs}{l_{av}})^2 \leq PoA \leq 1 + \frac{2cs}{l_{av}} + (\frac{cs}{l_{av}})^2$$

■

The price of anarchy depends on the average load on the server and on the network delay. For the more general case, in Section VI-C we present the estimations derived from simulations.

VI. Simulation Experiments

In this section we show the results from the two groups of experiments. First, we investigate convergence time of the distributed algorithm. Second, we assess the loss of performance in an organizationally-distributed system compared to the optimal, central solution. The loss is computed as a ratio of the total processing times.

A. Settings

We experimented on two kinds of networks: homogeneous, with equal communication latencies (c_{ij} = 20); and heterogeneous, where latencies were based on measurements between PlanetLab nodes[1] expressed in milliseconds[2].

In the initial experiments, we analyzed networks composed of 20, 30, 50, 100, 200 and 300 serves. We also performed some experiments on larger networks (500, 1000, 2000, 3000 servers). The processing speeds of the servers s_i were uniformly distributed on the interval $\langle 1,5\rangle$.

We conducted the experiments for exponential and uniform distribution of the initial load over the servers. For each distribution we analyzed five cases with the average load equal to 10, 20, 50, 200 and 1000 requests (assuming that processing a single request on a single server takes 1ms). We also analyzed the case of peak distribution – with 100.000 requests owned by a single server.

We evaluated the result based on the distance to the optimal solution, which because of the $O(m^6)$ complexity

[1] http://iplane.cs.washington.edu/data/data.html

[2] The dataset does not contain latencies for all pairs of nodes, so we had to complement the data by calculating minimal distances.

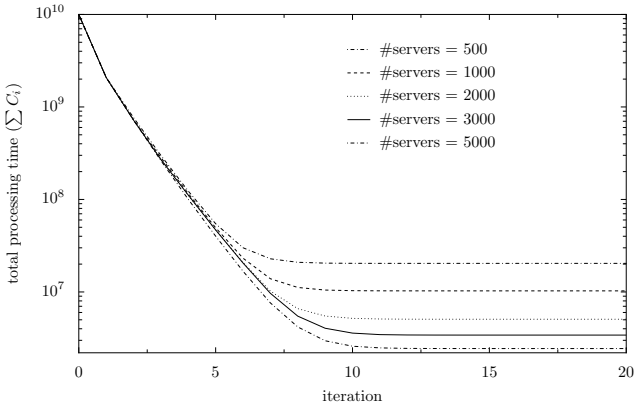

Figure 2: The convergence of the distributed algorithm for peak distribution of initial loads.

of standard solvers (see Section III) was approximated by our distributed algorithm.

B. Convergence time of the distributed algorithm

In the first series of experiments, we evaluated the efficiency of the distributed algorithm measured as the number of iterations the algorithm must perform in order to decrease the difference between the total processing times in the current and the optimal requests distributions to less than 2% of the average load. In a single iteration of the distributed algorithm, each server executes Algorithm 2; if there were many pairs of the servers to be optimized we run optimization in the random order. Table I summarizes the results.

The results indicate that the number of iterations mostly depends on the size of the network and on the distribution of the initial load. The type of the network (planet-lab vs. homogeneous) does not influence the convergence time. Larger networks and peak distribution result in higher convergence times. In all considered networks, the algorithm converged in at most 9 iterations.

Next, we decreased the required precision error from 2% to 0.1%, and ran the same experiments. The results are given in Table II. In this case, similarly, the required number of iterations was the highest for peak distribution of the initial load. In each case the algorithm converged in at most 11 iterations. Even for 300 servers the average number of iterations is below 8. Also, the standard deviations are low, which indicates that the algorithm is stable with respect to its fast convergence.

Also, we assessed whether a variation of the distributed algorithm that does not eliminate negative cycles (Appendix A) has a slower convergence time. Although required to prove the convergence (Section IV-B), eliminating the negative cycles is complex in implementation and dominates the execution time.

We compared two versions of the distributed algorithm: without negative cycle removal; and with the removal every two iterations of the algorithm. The number of iterations for two versions of the algorithm were *exactly* the same

		# iterations		
		average	max	st. dev.
$m \leq 50$	uniform	1.65	3	0.49
	exp.	2.35	3	0.47
	peak	4.87	6	0.71
$m = 100$	uniform	2.0	2.0	0.0
	exp.	2.62	3	0.48
	peak	6.88	7	0.32
$m = 200$	uniform	2.1	3	0.33
	exp.	3.1	4	0.33
	peak	7.84	8	0.37
$m = 300$	uniform	2.0	2	0.0
	exp.	3.25	4	0.43
	peak	8.0	8	0.0

Table I: The number of iterations of the distributed algorithm required to obtain at most 2% relative error in the total processing time ΣC_i.

		# iterations		
		average	max	st. dev.
$m \leq 50$	uniform	5.1	7	1.0
	exp	5.5	7	0.9
	peak	6.4	7	0.5
$m = 100$	uniform	5.8	9	1.6
	exp.	6.3	9	1.5
	peak	8.0	9	0.2
$m = 200$	uniform	6.1	9	2.2
	exp.	7.1	10	2.0
	peak	9.9	10	0.3
$m = 300$	uniform	6.2	10	2.4
	exp.	7.7	11	2.0
	peak	10.0	10	0.0

Table II: The number of iterations of the distributed algorithm required to obtain at most 0.1% relative error in the total processing time ΣC_i.

in all 6000 experiments. These result show that the cycles which happen in practice can be efficiently removed by pure Algorithm 1. Also, the negative cycles are rare in practice.

Finally, we analyzed the convergence of the distributed algorithm without negative cycles elimination on larger networks (Figure 2). The previous experiments shown that the algorithm convergence is the slowest for peak distribution of the initial load, therefore we chose this case for the analysis. The experiments used heterogeneous network. The results indicate that even for larger networks the total processing time decreases exponentially.

C. Cost of selfishness

In the second series of experiments we experimentally measured the cost of selfishness as the ratio between total processing times in cases of selfish and cooperative servers (Table III). In each experiment, the Nash equilibrium was approximated by the following heuristics. Each server was playing its best response to the current distribution of requests. We terminated when all servers in two consecutive steps changed the distribution of their requests by less than 1%. We computed the ratio of the total processing times: the (approximated) Nash equilibrium to the optimal value.

The cost of selfishness is low. The average is below 1.06; and the maximal value is below 1.15. The estimation

			Ratio		
			avg.	max	st. dev.
const s_i	$l_{av} \leq 30$	c_{ij} = 20	1.041	1.098	0.029
		PL	1.014	1.049	0.007
	l_{av} = 50	c_{ij} = 20	1.114	1.150	0.031
		PL	1.011	1.033	0.006
	$l_{av} \geq 200$	c_{ij} = 20	1.024	1.055	0.018
		PL	1.003	1.022	0.003
uniform s_i	$l_{av} \leq 30$	c_{ij} = 20	1.000	1.022	0.001
		PL	1.000	1.000	0.000
	l_{av} = 50	c_{ij} = 20	1.041	1.062	0.018
		PL	1.000	1.000	0.000
	$l_{av} \geq 200$	c_{ij} = 20	1.001	1.029	0.006
		PL	1.000	1.000	0.000

Table III: Experimental assessment of the cost of selfishness: ratios between total processing times in cases of selfish and cooperative servers.

of the cost of selfishness is higher in case of constant processing rates s_i. It additionally depends on the ratio between the average initial load and the network latency and on the structure of the network. The highest cost is for homogeneous networks with constant processing rates and having medium initial load about 2 times longer than the mean communication delay. The experiments show that the cost of selfishness is independent of the size of the network and the type of distribution of initial loads.

VII. Extension: requests of different processing times; replication

Up to this point, we modeled a distributed request processing system, in which requests have the same size. In this section we show how our results extend to the model where the individual requests (constituting the load) have different durations and where the requests additionally have redundancy requirements (These extensions are particularly relevant for the problem of finding the replica placement in CDNs – here different data pieces have different popularities and data redundancy is a common requirement for increasing the availability).

We introduce the following additional notation. A task is an individual request. $J_i = \{J_i(k)\}$ denotes the set of tasks of organization i; $p_i(k)$ is the size (processing time) of the task $J_i(k)$.

First, let us analyze a problem in which the tasks have no redundancy requirements, i.e. each task has to be processed on exactly one server.

In order to find the optimal solution in this extended model, we start with solving the original problem (as defined in Section II) with $n_i = \sum_k p_i(k)$. In order to derive the actual distribution of the tasks, we discretize the fractions ρ_{ij} as follows. i should relay to j such subset $S_i(j) \subseteq J_i$ of its own tasks, so that the total error $\Sigma err(S_i(j))$ is minimized:

$$err(S_i(j)) = |\sum_{k:J_i(k)\in S_i(j)} p_i(k) - \rho_{ij} n_i|.$$

The rounding problem is the multiple subset problem with different knapsack capacities [10]. The problem is NP-complete but has a polynomial approximation algorithm.

Now consider a problem in which each organization must execute at least R copies of each task; each copy of the task should be executed at a different location (the execution of the tasks is replicated). This setting models a CDN, but also job processing, where to increase survivability important jobs are replicated on different partions of a datacenter or on different datacenters.

In this extended problem we have to introduce additional constraint on the fractions ρ_{ij} for the original problem (Section II): $\forall_{i,j}\, \rho_{ij} \leq \frac{1}{R}$, which guarantees that $R\rho_{ij} \leq 1$. With this constraint we can interpret $R\rho_{ij}$ as the probability of placing a copy of $J_i(k)$ at j; here the expected number of copies of $J_i(k)$ is $\sum_j R\rho ij = R$.

VIII. Related work

The congestion games [14], [23], [26], [31] define the model for analyzing the selfish behavior of users competing for commonly available resources. Similarly to our model, the cost of a particular resource is linearly proportional to the number of competitors using the resource. In contrast, our model more closely describes the cost of using a resource which depends also on the communication delay.

The assumptions in our model are similar as in the literature on network virtualization [5]. However in network virtualization the problems regard locating services which is different from optimizing the quality of serving the common user requests. The complexity of the solutions depend on the number of configurations (which here is unbounded) thus the solutions cannot be applied to our model.

The continuous allocation of requests to servers in our model is analogous to the divisible load model [19] with constant-cost communication (a special case of the affine cost model [6]) and multiple sources (multiple loads to be handled, [16], [36]). The main difference is the optimization goal: makespan is usually optimized in the divisible load model; in contrast, we optimize the average processing time, which, we believe, better models situations in which the load is composed of multiple, small requests issued by various users (the difference is analogous to $C_{\max}$ versus ΣC_i debate in the classic multiprocessor job scheduling). The other difference is how the network topology is modelled. The divisible load theory typically studies datacenter-type systems, in which the network topology is known and is a limiting factor, thus the transmissions must be scheduled in a similar way to the computations.

Distributed algorithms for load balancing mostly relay on local optimization techniques (see [1], [7], [38]). One of the most popular techniques is diffusive load balancing, similar in spirit to our distributed algorithm (see [2] and the references inside for the current state of the art and [38] for the basic mechanism description). These solutions, however, disregard the geographic distribution of the servers. Our

algorithm uses different idea – the diffusive process is used for calculating the relay fractions instead of for balancing the load. As the result, our local balancing must take into account different latencies between the servers which requires more subtle exchange mechanisms (Algorithms 1 and 2).

Our game-theoretic approach is comparable to the selfish job model [37]: the jobs independently chose the processor on which to execute. While some studies consider mixed case equilibria (making the model continuous similarly to ours), our model considers also communication latency. The common infrastructure models tend to have a low price of anarchy (of order $\log m / \log\log m$ [37]) — the low price of anarchy in our model extends these results.

Content delivery networks are one of the motivations for our model. Large companies, like Akamai, specialize in delivering the content of their customers so that the end users experience the best quality of service. Akamai's architecture is based on DNS redirections [24], [25], [34]. However, the description of the algorithms optimizing replica placement and request handling are not disclosed. Still, Akamai's infrastructure is owned and controlled by a single entity (Akamai), thus they do not need to solve the game-theoretic equivalent of our model.

CoralCDN [18] is a p2p CDN consisting of users voluntarily devoting their bandwidth and storage to redistribute the content. In CoralCDN the popular content is replicated among multiple servers (which can be viewed as relaying the requests); the requests for content are relayed only between the servers with constrained pairwise RTTs (which ensures the proximity of delivering server). Our mathematical model formalizes the intuitions behind heuristics in CoralCDN.

[13] shows a CDN based on a DHT and heuristic algorithms to minimize the total processing time. Although each server has a fixed constrains on its load/bandwidth/storage capacity, the paper does not consider the relation between server load and its performance degradation. The evaluation is based on simulation; no theoretical results are included.

The problem of mirroring in the Internet is analyzed in [15], [30]. Both papers show different approaches to choosing locations for replicas so that the average network delay between data locations and end-users is minimized. The impact of servers' congestion is not taken into consideration.

IX. Conclusions

In this paper we present and analyze a model of a distributed system that minimizes the request processing time by distributing the requests among servers. Existing models assume that the processing time is dominated either by the network communication delay or by congestion of servers. In contrast, *in our model, the observed latency is the sum of the two delays: network delay and congestion*. Our model can be used in different kinds of problems in distributed systems, ranging from routing in content delivery networks to load balancing in a cloud of servers.

We show that the problem of minimizing the total processing time can be stated as an optimization problem in the continuous domain. We prove that the optimization problem is convex and, in particular, polynomially solvable. We propose a distributed algorithm that, according to our experimental evaluation, even in a network consisting of thousands of servers requires only a dozen of messages (not counting the gossiping to exchange the information) sent by each server to converge to a solution worse than at most 0.1% of the optimum. We present the properties of the distributed algorithm. We show how to estimate the distance between the current solution found by the algorithm and the optimal solution. The estimation requires solving the subproblem of finding the maximal flow of the minimal cost in a graph. However, the distributed algorithm still outperforms standard optimization techniques. Based on the experiments, we argue that in practice this part of the algorithm can be omitted, as it does not influence the algorithm efficiency.

We also analyze how the lack of coordination influences the total processing time. We give theoretical bounds for the price of anarchy for homogeneous networks and high average loads. Additionally, we assess the price of anarchy experimentally on heterogenous networks. In both cases the price of anarchy is low ($1 + \frac{2cs}{l_{av}}$ in the theoretical analysis, and below 1.15 in the experiments).

We show how the problem can be extended and, in particular, how it can be applied to find the optimal distribution of the data (even with their replicas) when the access pattern of the users is known, and how it can be applied to the case when the jobs have different processing times.

Our results — the low price of anarchy and an efficient distributed optimization algorithm — indicate that a fully distributed query processing system can be efficient. Thus, instead of buying services from dedicated cloud providers or CDN operators, smaller organizations, such as ISPs or universities, can gather in consortia effectively serving the participators' needs.

Acknowledgements The authors were supported by the Fundation for Polish Science "Homing Plus" Programme (grant no. HOMING PLUS/2010-2/13) co-financed by the European Regional Development Fund (Innovative Economy Operational Programme 2007-2013) and by EU's Human Capital Program "National PhD Programme in Mathematical Sciences" carried out at the University of Warsaw. Authors acknowledge helpful discussions with Maciej Drozdowski.

References

[1] H. Ackermann, S. Fischer, M. Hoefer, and M. Schöngens. Distributed algorithms for qos load balancing. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 197–203, 2009.

[2] C. P. J. Adolphs and P. Berenbrink. Improved bounds for discrete diffusive load balancing. In *IPDPS*, pages 820–826. IEEE Computer Society, 2012.

[3] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[4] J. Allard, S. Cotin, F. Faure, P. Bensoussan, F. Poyer, C. Duriez, H. Delingette, L. Grisoni, et al. Sofa-an open source framework for medical simulation. In *Medicine Meets Virtual Reality, MMVR 15*, 2007.

[5] D. Arora, A. Feldmann, G. Schaffrath, and S. Schmid. On the benefit of virtualization: Strategies for flexible server allocation. In *Proceedings of USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE '11)*, 2011.

[6] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, and Y. Yang. Scheduling divisible loads on star and tree networks: results and open problems. *Parallel and Distributed Systems, IEEE Transactions on*, 16(3):207–218, 2005.

[7] P. Berenbrink, M. Hoefer, and T. Sauerwald. Distributed selfish load balancing on networks. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 1487–1497, 2011.

[8] D. P. Bertsekas. Auction algorithms for network flow problems: A tutorial introduction. *Computational Optimization and Applications*, 1:7–66, 1992.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Prentice-Hall, 1989.

[10] A. Caprara, H. Kellerer, and U. Pferschy. A PTAS for the multiple subset sum problem with different knapsack capacities. *Information Processing Letters*, 73(3-4), 2000.

[11] E. Chan-Tin and N. Hopper. Accurate and provably secure latency estimation with treeple. In *NDSS*. The Internet Society, 2011.

[12] A. Chawla, B. Reed, K. Juhnke, and G. Syed. Semantics of caching with spoca: a stateless, proportional, optimally-consistent addressing algorithm. In *USENIXATC*, 2011.

[13] Y. Chen, R. H. Katz, and J. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *IPTPS, Proceedigs*, pages 306–318, London, UK, 2002. Springer-Verlag.

[14] G. Christodoulou and E. Koutsoupias. The price of anarchy of finite congestion games. In *STOC, Proceedings*, pages 67–73, 2005.

[15] E. Cronin, S. Jamin, C. Jin, A. R. Kurc, D. Raz, Y. Shavitt, and S. Member. Constrained mirror placement on the internet. In *JSAC*, pages 31–40, 2002.

[16] M. Drozdowski and M. Lawenda. Scheduling multiple divisible loads in homogeneous star systems. *Journal of Scheduling*, 11(5):347–356, 2008.

[17] P. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *IPDPS, Proc.*, pages 125–132. ACM, 2004.

[18] M. Freedman. Experiences with coralcdn: A five-year operational view. In *NSDI USENIX, Proceedings*, 2010.

[19] M. Gallet, Y. Robert, and F. Vivien. Divisible load scheduling. In Y. Robert and F. Vivien, editors, *Introduction to Scheduling*. CRC Press, Inc., 2009.

[20] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15:430–466, July 1990.

[21] D. Goldfarb and S. Liu. An o(n3) primal interior point algorithm for convex quadratic programming. *Math. Program.*, 49:325–340, January 1991.

[22] D. S. Hochbaum. Complexity and algorithms for nonlinear optimization problems. *Annals OR*, 153(1):257–296, 2007.

[23] E. Koutsoupias and C. H. Papadimitriou. Worst-case equilibria. *Computer Science Review*, 3(2):65–69, 2009.

[24] F. Leighton and D. Lewin. Global hosting system. US Patent No. 6,108,703.

[25] R. Mahajan. How akamai works? http://research.microsoft.com/en-us/um/people/ratul/akamai.html.

[26] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani. *Algorithmic Game Theory*, chapter Routing Games. Cambridge University Press, 2007.

[27] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44:2–19, August 2010.

[28] B. D. P. *Network Optimization: Continuous and Discrete Models*. Athena Scientific, 1998.

[29] G. Pallis and A. Vakali. Content delivery networks. *Communications of the ACM*, 49(1):101, 2006.

[30] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *IEEE INFOCOM, Proceedings*, pages 1587–1596, 2001.

[31] A. Roth. The price of malice in linear congestion games. In *WINE, Proceedings*, pages 118–125, 2008.

[32] K. Rzadca, A. Datta, and S. Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. In *ICDCS, Proceedings*, pages 599–609, 2010.

[33] P. Skowron and K. Rzadca. Network delay-aware load balancing in selfish and cooperative distributed systems. *CoRR*, abs/1212.0421, 2012.

[34] A.-J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind Akamai. *SIGCOMM Comput. Commun. Rev.*, 36:435–446, August 2006.

[35] M. Szymaniak, G. Pierre, and M. Steen. Scalable cooperative latency estimation. In *ICPADS*, 2004.

[36] B. Veeravalli and G. Barlas. Efficient scheduling strategies for processing multiple divisible loads on bus networks. *Journal of Parallel and Distributed Computing*, 62(1):132–151, 2002.

[37] B. Vocking. Selfish load balancing. In N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, editors, *Algorithmic Game Theory*. Cambridge University Press, 2007.

[38] C. Xu and F. C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluw. Acad. Pub., 1997.

APPENDIX
REMOVING NEGATIVE CYCLES

The problem of negative cycles removal can be reduced to finding the maximal flow of the minimal cost in a graph. The problem of finding the maximal flow of the minimum cost is well studied in the literature; there are many algorithms [3], [28]. In particular the auction algorithms [8] , ε-relaxation method [9], and the approximation method for finding minimum circulation [20] are the examples of the distributed algorithms solving the problem.

For the purpose of proving the reduction we introduce the following notation. $out(\rho',i)$ denotes the total amount of requests that in a partial solution ρ' are relied by a server i to all other servers: $out(\rho',i) = \sum_{j\neq i} r_{ij}$. $in(\rho',i)$ denotes the total amount of requests that in ρ' are relied by all other servers to i, $in(\rho',i) = \sum_{j\neq i} r_{ji}$.

We construct the max-flow min-cost graph as follows. For each server i we introduce two graph vertices: the front i_f and the back i_b. There are two additional vertices: s (source) and t (target). The source s is linked with each front node, i_f with an edge (s,i_f) with zero cost and capacity equal to $out(\rho',i)$. Each back node, i_b is linked with the target t with an edge (i_b,t) with zero cost and capacity equal to $in(\rho',i_b)$. Intuitively, capacity of (s,i_f) equal to $out(\rho',i)$ represents a server i sending out the same load $out(\rho',i)$ as in ρ'; conversely, capacity of (i_b,t) equal to $in(\rho',i)$ represents server i accepting the same amount of load as in ρ'.

There are also edges between front and back nodes: for each pair (i_f, j_b), $i \neq j$ there is an edge with cost equal to c_{ij} and infinite capacity.

The maximal flow of the minimal cost f between s and t can be mapped to a new partial solution ρ'': a flow on an edge $(i_f,\ j_b)$ f_{ij} corresponds to server i relying f_{ij} of its own requests to server j. Observe that, as capacity $(s,i_f) = out(\rho',i)$ and capacity $(i_b,t) = in(\rho',i)$, the load of i-th server in ρ'' is equal to its load in ρ. Additionally, there are no negative cycles in ρ: a negative cycle would result in a non-minimal cost of the flow f.

APPENDIX
VALIDATION OF THE CONSTANT LATENCY

We experimentally verified how the amount of the load sent over the network influences the communication delay between the servers. We randomly selected 60 PlanetLab servers, scattered around Europe, and simulated different intensity of the background load in the following way. Each server choses its 5 neighbors randomly , but in a way that each server has exactly 5 neighbors. Then the servers start sending data with constant throughput to its 5 neighbors. In different experiments, we used 8 values of the throughputs: 10KB/s, 20KB/s, 50KB/s, 100KB/s, 200KB/s, 500KB/s, 1MB/s, 2MB/s. If a particular throughput was not achievable, the server was just sending data with the maximal achievable throughput. By the nature of experiments on PlanetLab, we were not granted a dedicated access to the machines; thus other experiments running on the same servers added further, unknown network transfers. Additionally, almost all PlanetLab servers do not specify the bandwidth of their Internet connection, nor the historical bandwidth usage. For each value of the background load we calculated the average round trip time (RTT) between the server and each of its 5 neighbors (we used the average from 300 RTT samples).

t_b	$e(\cdot,\cdot,t_b)$	
	μ	σ
10 KB/s	0.0	0.0
20 KB/s	-0.05	0.21
50 KB/s	-0.05	0.27
0.1 MB/s	-0.08	0.33
0.2 MB/s	0.0	0.37
0.5 MB/s	0.28	0.8
2 MB/s	0.45	1.31
5 MB/s	0.18	0.8

Table IV: The relative deviation of the average throughput caused by the increase of the background load (after removal of 5% largest deviations).

Let $rtt(s_i,s_j,t_b)$ denote the average rtt between servers s_i and s_j with the background load generated with throughput t_b. For each pair of the servers s_i and s_j for which we measured the RTT, and for each value of the background throughput t_b we calculated the relative deviation of the average throughput caused by the increase of the background load compared to the minimal throughput 10KB/s: $e(s_i,s_j,b_t) = \frac{rtt(s_i,s_j,t_b) - rtt(s_i,s_j,10KB/s)}{rtt(s_i,s_j,10KB/s)}$. For each value of the background throughput, we removed 5% of the largest deviations and then calculated the mean from deviations $e(s_i,s_j,b_t)$, averaged over all pairs of servers (μ). For each value of the background throughput we additionally calculated the standard deviations (σ). These results are presented in Table IV.

From the data we see that up to $b_t = 0.2$MB/s, which corresponds to the case where each server accepts $5 \cdot 0.2 \cdot 8 =$ 8Mb/s of incoming data, the average RTT was not influenced by the background throughput. This is also confirmed by the statistical analysis of the data run for the RTTs (instead of for deviations). For $b_t \leq 0.2$MB/s the ANOVA test (which we run for the whole population – without removing 5% of the highest RTTs) confirmed the lack of dependency null hypothesis (that the background throughput does not influence the RTTs) for over 56% of the pairs of servers. For $b_t \leq 0.1$MB/s (corresponding to 4Mb/s of incoming throughput) the ANOVA test confirmed null hypothesis for over 70% of the pairs of servers and for $b_t \leq 50$KB/s (corresponding to 2Mb/s of incoming throughput) for over 90% of the pairs. We consider that these results strongly justify the assumption of a constant latency in our model.