An Experimental Study of k-Splittable Scheduling for DNS-Based Traffic Allocation

Amit Agarwal[⋆], Tarun Agarwal[⋆], Sumit Chopra[⋆⋆], Anja Feldmann[⋆⋆⋆], Nils Kammenhuber[⋆⋆⋆], Piotr Krysta[†], and Berthold Vöcking[‡]

Abstract. The Internet domain name service (DNS) uses rotation of address lists to perform load distribution among replicated servers. We model this kind of load balancing mechanism in form of a set of request streams with different rates that should be mapped to a set of servers. Rotating a list of length k corresponds to breaking streams into k equally sized pieces. We compare this and other strategies of how to break the streams into a bounded number of pieces and how to map these pieces to the servers.

One of the strategies that we study computes an *optimal k-splittable allocation* using a scheduling algorithm that breaks streams into at most $k \geq 2$ pieces of possibly different size and maps these pieces to the servers in such a way that the maximum load over all servers is minimized. For this purpose we use a recently introduced algorithm for the k-splittable machine scheduling problem. The running time of this algorithm is only linear in the number of streams but exponential in the number of machines. We suggest an improvement to this algorithm for the case of identical machines (servers), and prove that our modification reduces the running time significantly. This enables us to compute optimal allocations for several hundreds of servers, although this problem is known to be NP-hard.

Our experimental study is done using the network simulator SSFNet. We study the average and maximum delay experienced by HTTP requests for various traffic allocation policies and traffic patterns. Our simulations show that splitting data streams can reduce the maximum as well as the average latency of HTTP requests significantly. This improvement can be observed even if streams are simply broken into k equally sized pieces that are mapped randomly to the servers. Using allocations computed by machine scheduling algorithms, we observe further significant improvements.

[⋆] IIT Delhi, India. This work was done while the author was visiting the MPI in Saarbrücken.

[⋆⋆] Department of Computer Science, Hansraj College, University of Delhi, India. This work was done while this author was visiting the MPI in Saarbrücken.

[⋆⋆⋆] Department of Computer Science, Technische Universität München, Germany.

[†] Max-Planck-Institut für Informatik, Saarbrücken, Germany.

[‡] Department of Computer Science, Universität Dortmund, Germany.

1 Introduction

The Internet's Domain Name System (DNS) is responsible for resolving URLs like "`www.uni-dortmund.de`" into IP addresses. Increasingly, this service is also being used to perform load distribution among distributed Web servers. Busy sites are replicated over multiple servers, with each server running on a different end system, and each having a different IP address. Thus a set of IP addresses is associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query to such an address, the server returns the entire set of addresses but rotates the ordering of addresses within each reply. Because a client typically sends its HTTP request message to the address listed first, the traffic is distributed among all replicated servers (cf. [7]).

One can view the requests that are directed to the same URL as traffic streams. The rotation of address lists basically splits these streams into equally sized pieces. Suppose the request streams are formed by a sufficiently large number of clients. Then the arrivals of requests can be described by a stochastic process, e.g., a Poisson process. In the scenario that we consider there are n streams and m identical servers. Let λ_j denote the rate of stream $j \in [n]$, i.e., the expected number of requests in some specified time interval. Under this assumption, rotating a list of k servers corresponds to splitting stream j into k substreams each of which having rate λ_j/k. The following slightly more sophisticated stochastic splitting policy would possibly achieve a better load balancing. Suppose, the DNS attaches a vector $p_1^j, \ldots, p_k^j$ with $\sum_i p_i^j = 1$ to the list of each stream j. In this way, every individual request in stream j can be directed to the ith server on this list with probability p_i^j. This policy breaks Poisson stream j into k Poisson streams of rate $p_1^j\lambda_j, \ldots, p_k^j\lambda_j$, respectively.

The focus of this paper does not lie on the issue of how exactly to implement such strategies within the DNS implementation. Instead, our goal is to study the impacts of different allocation strategies a few step ahead of current DNS implementations. In particular, we shall compare the following allocation strategies.

– simple random allocation, i.e., each stream is mapped as a whole to a randomly selected server;
– random k-split allocation, i.e., each stream is broken into k equally sized pieces each of which is assigned to a random server;

- LL heuristic k-split allocation, i.e., streams are broken into k equally sized pieces that are allocated to the servers using the well-known Least Loaded (LL) heuristic from machine scheduling;
- optimal k-split allocation, i.e., a scheduling algorithm computes the sizes of the pieces as well as the mapping of these pieces to the servers in such a way that the maximum load over all servers is minimized.

The optimal k-split allocation is computed using a variant of a recently presented algorithm for the k-splittable machine scheduling problem. Suppose a set of jobs $[n] = \{1, \ldots, n\}$ need to be scheduled on a set of machines $[m] = \{1, \ldots, m\}$. The jobs have the sizes $\lambda_1, \ldots, \lambda_n$. In this paper, we assume that the machines are identical. Every job can be broken into at most k pieces. We identify data streams with jobs and servers with machines. The objective is to find a mapping of the pieces of the jobs to the machines so that the makespan, i.e., the maximum load over all machines is minimized.

Scheduling with bounded splittability was introduced by Shachnai and Tamir [10]. They prove that the problem is NP-hard and present approximation algorithms. Krysta et al. [6] presented an exact algorithm for the k-splittable machine scheduling problem with running time $O(m^{m+m/k} + n)$. Thus the problem can be solved in polynomial time for any fixed number of machines. It is well known that the classical unsplittable scheduling problem is NP-hard already for two machines. Thus the result from [6] proves that bounded splittability reduces the complexity of the problem for a fixed number of machines.

At this point, we want to remark that the algorithms in [6, 10] do not only work for identical but also for uniformly related machines, i.e., for machines of different speeds. In this paper, we focus on the special case of identical machines. Furthermore, we want to remark that the above and all following bounds on the running time refer to algorithms that compute feasible allocations for any given upper bound on the makespan. These algorithms, in turn, can be used to compute the optimal makespan by applying binary search techniques.

When implementing the algorithm from [6], we learned that the exponential term in the running time let us only compute optimal assignments for relatively small numbers of machines. For some randomly generated instances with less than 40 machines the algorithm did not terminate in reasonable time. It was obvious, however, that, in the case of identical machines, one could exploit symmetries to speed up the algorithm. Clearly, because of the NP-hardness of the problem, the best one can hope for is to soften the exponential influence of the number of the machines. The theoretical contribution of this paper is an upper bound of

$$O\left((2m)^{m/(k-1)+1}(m/k)+n\right)$$

on the running time of an exact algorithm for the k-splittable scheduling problem on identical machines. Observe that this result yields the first complete tradeoff on the influence of different degrees of splittability on the running time ranging from polynomial time in the case of $k = \Omega(m)$ to exponential time when $k = O(1)$.

In Section 2, we present this improved algorithm and its analysis. Furthermore, we give some experimental results for different input distributions showing that the improved algorithm can be used to compute optimal solutions for several hundred machines very efficiently. In Section 3, we present our comparative study of different traffic allocation strategies applied to a simple network topology in which we measure average and maximum delays of HTTP requests. For a brief summary of these results, we refer the reader to the Conclusions presented in Section 4.

2 An improved algorithm for k-splittable scheduling

In this section, we present an improved algorithm for k-splittable scheduling on identical machines. Let $x_{i,j}$ denote the fraction of job j that shall go to machine i. We assume that a value z for the makespan (maximum load) is given and we are seeking for an algorithm that either computes an assignment satisfying $\max_{i\in[m]} \sum_{j\in[n]} \lambda_j x_{ij} \leq z$ or returns that there is no k-splittable assignment with maximum load z. Given such an algorithm, the original optimization problem can be solved in a polynomial number of iterations by applying binary search techniques over the rational numbers [5, 9]. The binary search works despite the possibility of irrational splits because the optimal solution can be proved to be rational [6].

First, let us describe the original algorithm from [6]. Then we will introduce a small modification to this algorithm that, however, leads to a dramatic reduction of the running time.

The original algorithm. The main difficulty in obtaining an algorithm for k-splittable problems is that there is an unbounded number of possible ways to cut a job into pieces. The analysis in [6], however, shows that any optimal assignment can be transformed into an assignment fulfilling certain conditions so that the search space for optimal cuts and assignments can be bounded in such a way that only $O(m^{m+\frac{m}{k-1}})$ assignments need to be taken into consideration.

In order to describe this algorithm, let us switch to a generalization of the k-splittable scheduling problem in which each job j comes with its own splittability $k_j \geq 2$. Initially $k_j = k$, for every $j \in [n]$. Let c_i denote the free capacity of machine i. Initially c_i ($i \in [m]$) corresponds to the given makespan z, but during the course of the algorithm these numbers are reduced appropriately whenever a new job is assigned to the corresponding machine. The algorithm consists of two phases, A and B.

Phase A is defined recursively. We pick exactly one job and assign a piece of this job to one of the machines. Let $j \in [n]$ denote the assigned job, let $\pi_j \leq \lambda_j$ denote the size of the assigned piece, and let $i \in [m]$ denote the machine to which we assign the job. Then, after we have assigned the piece of job j to machine i, we are left with a subproblem in which k_j is decreased by one and c_i as well as λ_j are decreased by π_j. This subproblem is solved recursively.

It remains to specify how to choose the first job, the size of the assigned piece, and the machine. The *bulkiness* of a job j is defined as $\lambda_j/(k_j - 1)$. The algorithm chooses the *bulkiest* job j and (using a backtracking approach) it tries all ways to choose a machine i and cut a piece from job j such that machine i is *saturated*, i.e., the capacity of this machine drops to zero. There is one exception from this rule: Since the splittability k_j of a job is decreased whenever a piece is cut off, a remaining piece can eventually become unsplittable. Since this remaining piece will be infinitely bulky, it will be scheduled next. In this case, all machines that can accommodate this piece need to be tried.

Phase B starts as soon as the bulkiest job j is too small to saturate any subset of $k_j - 1$ machines, that is when $\lambda_j/(k_j - 1) \leq \min\{c_i : i \in I\}$, where I is the set of the remaining machines with (residual) capacities c_i. In this case all remaining jobs can be assigned using a simple greedy heuristic known as McNaughton's rule [8].

Phase B takes only $O(m + n)$ time. Phase A can take much longer due to the backtracking over all possible choices for the machines. The depth of the search tree is bounded above by $m + \frac{m}{k-1}$ because each fully assigned job saturates at least $k - 1$ machines so that the number of machines with positive capacity is zero after at most $m + \frac{m}{k-1}$ rounds. The number of possible choices for the machines to which a job can be assigned is at most m. Consequently, the total size of the search space and, hence, the running time of phase A is at most $O(m^{m+\frac{m}{k-1}})$.

The difficult part of the analysis of this algorithm is its correctness. We refer the reader to [6] for a detailed proof.

The improved algorithm. We change the algorithm only in a small aspect, that is, we reduce the size of the search space by exploiting symmetries. One can consider all machines with the same remaining capacity as belonging to one equivalence class. Obviously, one does not need to backtrack over machines in the same class. In the considered case of identical machines, all *empty machines* (i.e., machines with remaining capacity z) fall into the same equivalent class. In contrast, the *partially filled machines* (i.e., machines with capacity in $(0, z)$) typically build equivalence classes of size one.

In order to simplify the implementation, we only exploit the symmetry among the empty machines. To be more precise, we pick jobs in the order of their bulkiness and for each job one of the following three cases is applied. Let j denote the index of the job being considered.

1) If $k_j \geq 2$ then we saturate one of the machines, selecting a machine from the class of the empty machines first and then trying all machines with reduced capacity.
2) If $k_j = 1$ then we put the remaining piece job j to a machine i with $c_i \geq \lambda_j$, first trying all partially filled machines and then one of the empty machines.
3) If $\frac{\lambda_j}{k_j - 1} \leq \min_{i\in[m]}(c_i)$ then Phase B (McNaughton) is initiated.

Obviously, our modification does not affect the correctness of the algorithm but it reduces the running time. One might object, however, that exploiting the symmetries cannot help too much as the assignment of jobs to machines destroys these symmetries rapidly. Nevertheless, the following theorem states that the running time is improved dramatically.

Theorem 1. *The running time of the improved algorithm for the k-splittable traffic allocation problem is $O\left((2m)^{m/(k-1)+1}(m/k) + n\right)$, for every $k \in \{2, \ldots, m\}$.*

Proof. At first, we investigate the size of the search tree that is explored within phase A. Observe that case 1 can be applied at most m times as each application of this case saturates one machine. Furthermore, case 2 can be applied to a job only after case 1 has been applied for $k - 1$

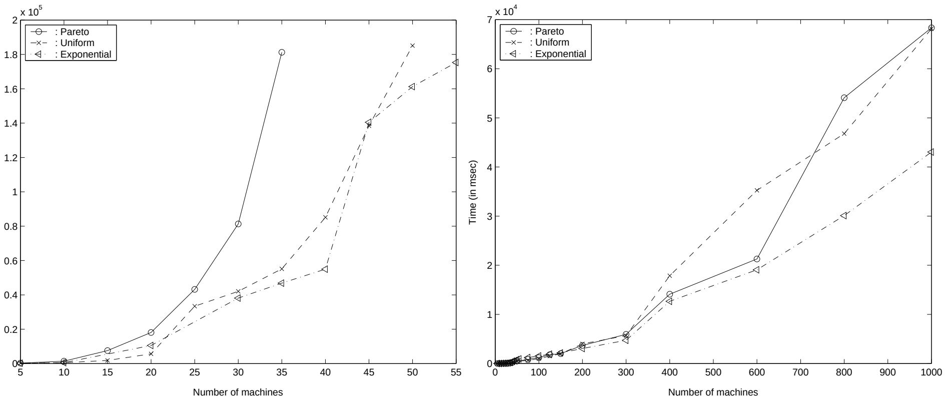

Fig. 1. The pictures show the running time of the original (left) and the modified algorithm (right) for three input distributions. The scale of the left picture is 10^6 msec, the scale of the right picture is 10^4 msec. The modified algorithm took less than 5 minutes for every instance for which it was tested. The original algorithm did not terminate in reasonable time for several instances. The average for this algorithm was taken only over those instances for which the algorithm terminated within 5 minutes.

times to the same job. Thus, the number of partially filled machines that can be generated in case 2 is at most $q = \left\lfloor \frac{m}{k-1} \right\rfloor$, and the number of rounds executed within phase A is bounded above by $\ell = \lfloor m + \frac{m}{k-1} \rfloor$.

With each path from the root to a leaf of this search tree, we can associate a 0-1 string of length ℓ as follows. A "0" at position i in this string means that one of the empty machines is chosen on level i of the search tree and a "1" means that a partially filled machine is chosen. If the length of the path is less than ℓ, then we add additional zeroes at the end of the string. The number of different ways to choose such a 0-1 string of length ℓ with at most q occurrences of "1" is $\sum_{p=0}^{q} \binom{\ell}{p} \leq (q+1)\binom{\ell}{q}$.

Now fix one of these 0-1 strings. This string might correspond to several paths in the search tree. Next we bound the number of these paths. If the string is fixed then the only freedom is to determine which of the partially filled machines to take on those tree levels marked with "1". The number of of different ways to choose these partially filled machines is at most $q!$ because q is an upper bound on the number of partially filled machines generated by the algorithm on any path and every allocation of a job to such a machine reduces this number by one. (More precisely, an allocation of a piece of a job due to case 1 to a partially filled machine decreases the number of partially filled machines existing at that time, and an allocation to a partially filled machine due to case 2 reduces the total number of partially filled machines generated on this path.) Thus the number of paths corresponding to any given 0-1 string is at most $q!$.

As a consequence, the number of paths and, hence, the number of leafs in the search tree is at most

$$\begin{aligned}(q+1)\binom{\ell}{q}q! &= (q+1)\,\frac{\ell!}{(\ell-q)!}\\ &\leq (q+1)\,\ell^{q}\\ &\leq \left(1+\frac{m}{k-1}\right)\left(m+\frac{m}{k-1}\right)^{m/(k-1)}\\ &\leq \left(1+\frac{m}{k-1}\right)(2m)^{m/(k-1)}\ .\end{aligned}$$

The number of nodes in the tree can be estimated by the number of leafs times $\ell = O(m)$, and each such node can be explored in constant time. Furthermore, the number of steps in the McNaughton phase is $O(m+n)$. Thus the overall running time is $O\left((2m)^{m/(k-1)+1}(m/k)+n\right)$. This completes the proof of Theorem 1.

Observe that the theorem yields a complete tradeoff on the influence of the different degrees of splittability on the running time. In particular, the following result was not known before.

Corollary 1. *For any constant $c > 0$ and $k \geq \max\{2, \frac{m}{c}\}$, k-splittable scheduling on identical machines is solvable in polynomial time.*

2.1 Experimental comparison of the two algorithms

We have implemented both the original algorithm and its modification in order to compare there running times. The number of jobs (streams) was chosen to be 50 times the number of machines (servers). This combination of the parameters generated the hardest instances for both algorithms, and it also seems to be a realistic setting for the investigated scenario of balancing Web traffic. The sizes of the jobs were generated according to different probability distributions: the uniform distribution, the exponential distribution, and the Pareto distribution with parameter $\alpha = 1.1$. The motivation for the latter choice is that similar distributions were observed in studies of Web traffic, see, e.g., [3]. The

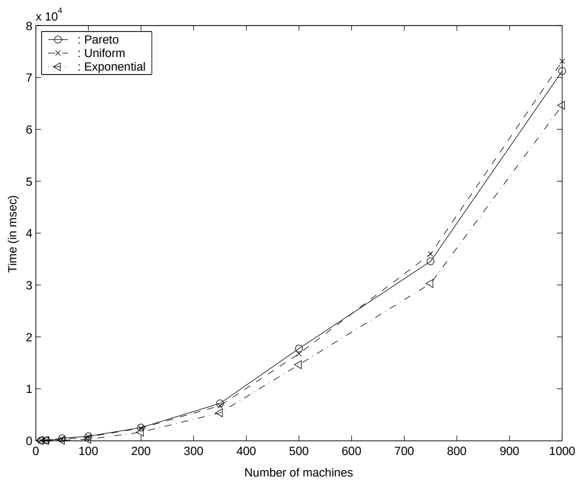

Fig. 2. The maximum running time of the modified algorithm observed in 100 instances.

plots of the running time for the different distributions are given in Figure 1. These times were obtained by averaging over 100 instances.

The plots show very clearly that the modified algorithm performs substantially better than the original one. In comparison to the original algorithm, the modified algorithm needs to backtrack only very rarely. In fact, most of the instances could be solved by the modified algorithm completely without backtracking. The Pareto distribution proved to generate the hardest instances. Other choices for the parameter α showed similar behavior. Under the exponential distribution both the algorithms perform best. The improved algorithm shows a very similar behavior for the Pareto and the uniform distribution. This can be explained by the fact that these two distributions produce a similar number of large jobs, i.e., jobs that cannot be immediately allocated by the McNaughton rule. In comparison, the exponential distribution produces only very few of these jobs. A further positive result is that the observed deviations in the running time of the modified algorithm is quite small. This is shown in a striking fashion by Figure 2.

3 Server load balancing results

3.1 A simple setup

Looking for a more sophisticated splitting scheme in order to improve load balancing in the server farm scenario, we implemented the four

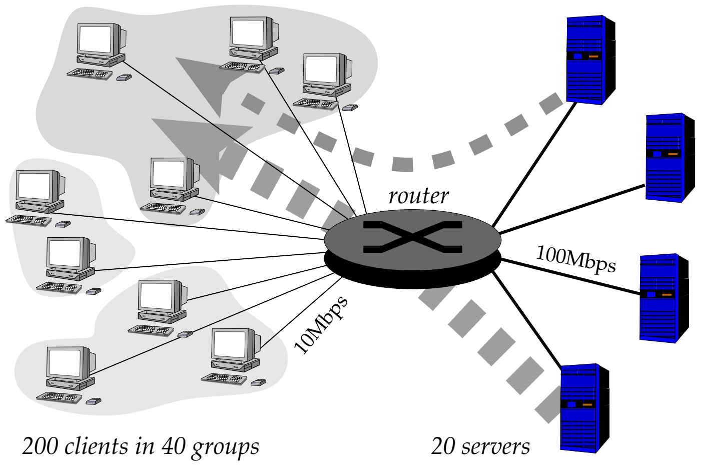

Fig. 3. The topology of the simulated Network model.

different strategies *simple random allocation*, *random k-split allocation*, *LL-heuristic k-split allocation*, and *optimal k-split allocation* as described in the Introduction.

These strategies were evaluated in simulations using the network simulator by Scalable Simulation Framework Research Network (SSFNet) version 1.4.0. We refer the reader to [11, 2] for details. The topology of the network simulated (shown in Fig. 3) consisted of n hosts (the Clients), a router with $n+m$ interfaces and another m hosts (the Servers). For each host on the client side, the bit rate of the interface (of the host-router link) was set to 10^7 bits/s with the link latency of 0.0. The bit-rates of the interfaces of the router and the buffer sizes were chosen sufficiently large to not to cause any bottleneck in the traffic flow. Bit rates of the hosts on the server side were set to 10^8 bits/s. For each host on the client side, the number of pages in a single session and the number of objects in a single page were set to 1. The inter-session time, inter-page time and inter-request time (time between two consecutive requests for different objects in a single page) were all set to 0. In this way, every host (on the Client side) sent out a more or less continuous request stream, and all hosts have the same rate. Now the rates λ_j's were realized by grouping these basically identical 200 request streams into 40 groups of possibly different sizes. Thus, 40 traffic streams (groups) were simulated using around 200 hosts (clients). The sizes of the groups and, hence, the rates $\lambda_1, \ldots, \lambda_{40}$ were generated according to different probability distributions. The number of servers was chosen to be 20.

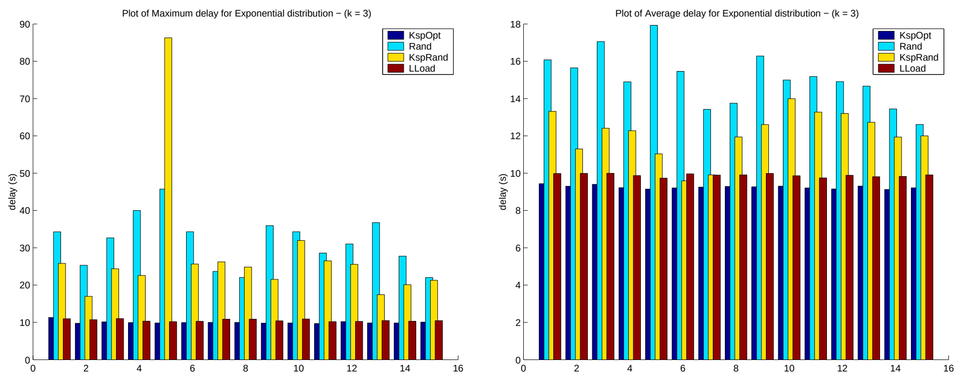

Fig. 4. Maximum and Average delay of the network when the group sizes are exponentially distributed (for 10 simulation runs).

The use of the probability vectors generated by the different allocation strategies was realized by associating a new probability attribute (p) with each client-server link in the traffic pattern [11, 2]. This was done by modifying the source code of SSFNet at the HTTP level. Using these probability vectors ,the traffic of each group was directed to the servers with the corresponding probability applied to individual requests. In this way, we obtained substreams of rates $p_1^j\lambda_j, \ldots, p_k^j\lambda_j$ as described in the Introduction.

Figure 4 describes the maximum and the average delays experienced by the network in 10 simulation runs, where the group sizes were exponentially distributed. By *delay*, we mean the time span between a client initiating a request and the same client receiving the last byte of the answer. We analyzed the delays as follows: First, the average delay within each group was computed. Then the maximum of these averages gave the maximum delay experienced by any group in the network and the average of these averages gave the average delay for all the groups in the network. These delays were measured after running the simulations for 1500 seconds, i. e., roughly 1500 sessions for each client.

A comparison between the various strategies clearly shows that the delays for HTTP requests when using the allocation of the optimal and heuristic k-split algorithm are much less as compared to the delays experienced when using the allocation of the randomized schemes. This improvement is larger when looking at the maximum rather than the average delay of HTTP requests. This is not surprising as both the heuristic as well as the optimal algorithm aim at minimizing the maximum load over all servers. In contrast, the random strategies produce an al-

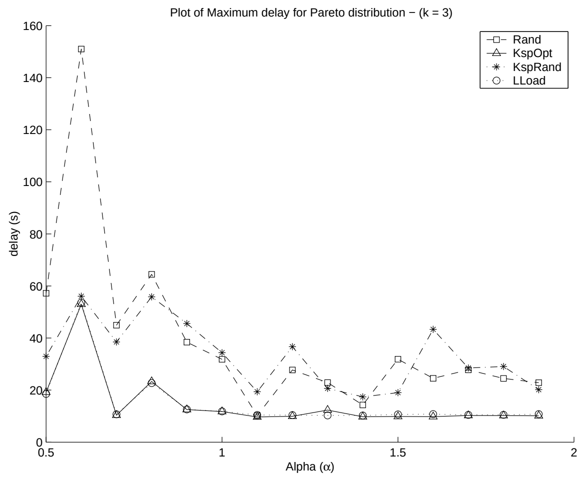

Fig. 5. Maximum delay of the network when the group sizes are Pareto distributed.

location in which a few of the servers are unlucky and receive a larger share of the load, which affects the maximum delay more than the average delay.

Besides to traffic patterns generated by the exponential distribution, we also tested patterns generated by the Pareto distribution with different choices for parameter α of this distribution, see Figure 5. We chose the Pareto distribution since it frequently shows up in various web traffic analyses as Zipf's law.

Not surprisingly, if α is small, then the experiments show quite large deviations. In principle, however, the results for the Pareto distribution are not significantly different from the ones for the exponential distribution. Again heuristic and optimal allocation are superior against the random allocations, and the measured differences between heuristic and optimal allocations are quite small.

Over all experiments we observed that splitting data streams into sub-
streams reduces the average as well as the maximum delays of HTTP requests significantly. We also studied different choices for the splittability parameter k. It turns out that setting $k = 3$ is a good choice. The results obtained for $k = 2$ are slightly worse. On the other hand, the improvements for $k \geq 4$ in comparison to $k = 3$ seem to be so small that they were completely covered by the stochastic variations in the randomly generated traffic patterns and other effects due to the complexity of the simulation environment.

3.2 Approaching Reality

Encouraged by our results with the rather simplistic simulation setup described above, we decided to move on to more realistic setups and check if our algorithm still performs well. Several aspects were targeted for improvements in further simulation runs:

– The delays between clients and servers are too small. In reality, the RTTs vary from as little as 0.1msec (clients and servers reside within the same LAN segment) up to several hundreds of milliseconds (e. g., client is in Europe and server is in New Zealand)
– The delays between clients and servers should be variable, as normally the clients are distributed across the world hence suffer different delays (see the two examples above)
– The existence of only a single router between clients and servers is unrealistic for most traffic.
– The assumption that all clients download files of exactly the same size again and again, without any delays between a download and its subsequent request, is extremely unrealistic as well. The research community has examined the statistical nature of WWW traffic, and the pattern of the traffic imposed by the clients on the servers should follow appropriate traffic models [3, 1].
– Having almost arbitrarily large queues in the routers is unrealistic. As router memory needs to be extremely fast, it is very expensive thus the queue length in a router is rather limited. Sooner or later this results in packets being dropped when the interface which the queue belongs to gets overloaded. These packet drops will cause the TCP (transmission control protocol) to retransmit the lost data and effect performance losses.

However, changing all these parameters in one single step seemed not to be a clever move. We think that it makes more sense instead to record at which points “nearer to reality” the algorithm performs how well compared to the three heuristics. The final goal was to test the algorithm and the three heuristics in a simulation setup where all the above aspects were addressed.

3.3 Increased link delays

Obviously, near-zero length transmission delays on the links result in a much quicker transmission time, especially for small files. During longer downloads, more possibilities arise that several TCP connections run in parallel, thus compete for bandwidth and therefore apply TCP's congestion control mechanism. Of course it would be interesting to see what role the congestion control can play. For the same reason, we chose to decrease the queue length of the routers, as dropped packets activate the congestion control mechanisms.

Since we wanted to study the effect of these changed network parameters, we decided to keep most the setup from the simulations so far—i. e. we grouped 200 uniform clients into 40 groups, and assigned vectors to each client containing the probabilities for accessing the 20 servers. The file sizes were set to 10 MBytes and the clients were instructed to issue a request as soon as the last one had completed, just as before.

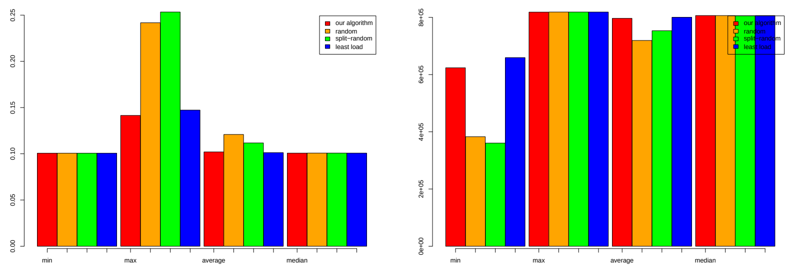

Fig. 6. Response delays (left) and download bandwidths (right) for the scenario with increased link delays. Pareto-distributed groups sizes, $\alpha = 1.0, k = 4$

Our algorithm also wins in the case of increased transmission delays. We refer to figure 6 as an example. When analyzing the time spans between a request being issued and the head of the answer being received, we can see that our algorithm wins over the other heuristics. This is also true when we look at the time span between the reception of the first byte and the last byte of the answer.

3.4 Using a realistic workload model

Next, we investigated how our algorithm would perform if we did not simulate artifical traffic where each client constantly downloads large files of equal sizes, but instead where the behaviour of a real human end user surfing the web is simulated.

It needs to be considered that if each of the clients follows a statistical model for the downloads, it does not impose a constant server load any more. Instead, it can be inactive for rather long periods of time. To offset this drop in the server load, the number of clients has to be increased. After doing some experiments, we chose to increase the number of clients to 2500 and to reduce the number of servers to 10.

Since now the file sizes were randomly distributed, the pure download times are not comparable as they were before. A kind of resort could be to calculate the download times for each file. Alas, this would discriminate agains small file sizes due to TCP's *slow start* mechanism, which arranges that a connection starts with a very low bandwidth and then adjusts to the available bandwidth over the course of the time. We offset this fact by ignoring connection data for objects smaller than 20 000 bytes ($20\times$ the chosen MSS—maximum segment size).

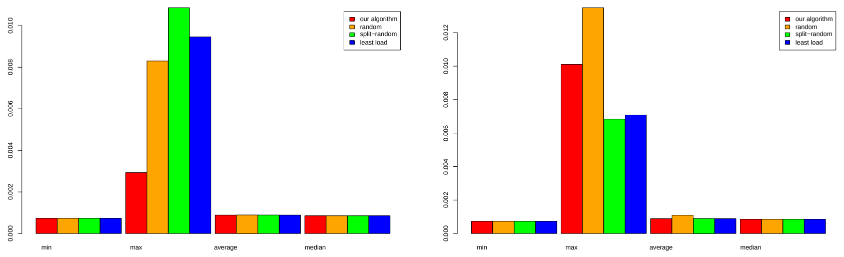

Fig. 7. Response delays for the web workload scenario. Pareto-distributed groups, $\alpha = 1.0$. Left side: $k = 4$, right side: $k = 2$. Note the high variability across the figures.

The simulation results we got were somewhat mixed. When comparing the bandwidths for our four balancing methods, we found almost no difference in the distribution of the bandwidths (although a very small advantage might perhaps be credited to our algorithm and the least load heuristic over the randomized schemes). At this point it needs to be stressed out that having a server farm of 10 servers for as little as 2 000 clients accessing web pages, albeit the only pages, is vastly overdimensioned. Certainly, an even greater number of clients would have been more realistic and favorable—however, a single SSFNet simulation run involving 2500 clients already needed 110 to 180 minutes of CPU time on a Sun Fire 880 and consumed 800 MBytes to 1.5 GBytes of RAM. Obviously, increasing the number of clients would have increased the consumption of these resources too much.

When comparing the maximum delays between the client issuing the request and the client receiving the first byte of the response, we get results that seem to give the credit for the best method again to our algorithm. Note, however, that the high statistical variability may shuffle around this order (see figure 7).

3.5 Small "Real world" simulations

The setup for the realistic simulations was chosen similar to the setup for the realistic web workload model: Again, we use 10 servers that are accessed by 2500 clients. However, the topology of the network is more complex in order to model reality.

The servers are connected to the same router, which represents the router of the server farm. This server-side router is connected over a single link of 2Gbit/sec to another router. This link represents the outbound connection of the provider owning the servers; the router at the other end represents its so-called *upstream provider.* [1]

As before, the clients are bundled into 50 groups. Each group represents an *autonomous system* (AS) [2] that hosts a number of clients accessing the servers of the server farm. Each group is simulated by a router that is connected to the router of the server farm's upstram provider via a 1 Gbps link. The link delays are randomised following an exponential distribution.

Over various simulation runs, we only can say that virtually no performance difference between the scheduling schemes can be observed. This holds regardless of the value for k (we investigated $2, 3, 4$ and 6) or the chosen distribution (equally sized jobs, exponentially distributed, uniformly distributed, or Pareto-distributed with $\alpha \in \{0.5, 1, 2.0\}$).

3.6 Large "Real world" simulations

We wanted to verify our suspicion that this result should mainly be attributed to the lack of workload imposed by too little clients on too many servers. Thus we increased the client-to-server ratio dramatically. We created two network scenarios with the same topology as above,

[1] An upstream provider is an ISP that connects its customers, which may be ISPs themselves, to the rest of the internet (perhaps via another upstream provider).

[2] The Internet is grouped into autonomous systems that have the sole administrative power over their own parts of the network.

with the exception that now we increased the number of clients to 8 000, the number of groups to 100, and reduced the number of servers even further down to 8. We used $k = 3$ and had the group sizes Pareto-distributed with $\alpha = 1.0$.

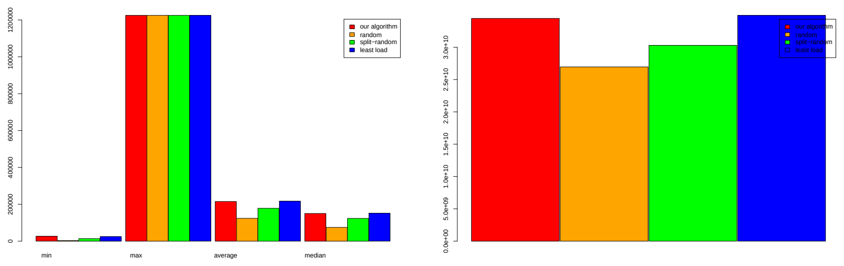

Fig. 8. Bandwidth minimum, maximum, average and median (left picture) and bandwidth variance for 8000 clients, 8 servers, with a “real-world” setup

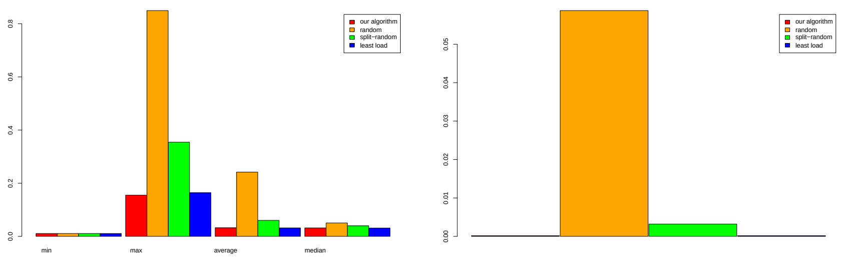

Fig. 9. Minimum, maximum, average delay and median (left picture) and delay variance for 8000 clients, 8 servers, with a “real-world” setup

Due to the extremely long simulation run time (over 12 hours for a simulated time of 1 hour; just the simulation initialization phase took around 10 minutes) and the extremely high RAM consumption (more than 2 GBytes), we only had resources to do a comparison based on just two simulation runs for each assignment method.

Finally we got the picture-book results that we were hoping for (see figures 8 and 9): Here our algorithm clearly wins over the heuristic methods. When comparing the medians of the average bandwidth achieved, it clearly declasses even the least-load scheme. However, more simulation should be performed on this before we can safely rule out the possibility of this being nothing but a statistical fluctuation.

4 Conclusions

We studied average and maximum delay experienced by HTTP requests for various traffic allocation policies and traffic patterns. Our simulation results show that splitting data streams into a small number of pieces can reduce the maximum as well as the average latency of HTTP requests significantly. This improvement is obvious even if streams are simply broken into k equally sized pieces that are simply mapped to the servers at random.

Besides to random allocations we devised and studied allocation strategies that carefully allocate pieces of jobs to machines with the objective to minimize the maximum load over all servers. These strategies lead to a further clear improvement in the maximum as well as the average latency of HTTP requests. It turned out, however, that the differences between the latencies obtained by the simple LL heuristic and the latencies obtained by an optimal k-splittable allocation are not significant.

There are several questions left open by our analysis. Certainly, some more test runs with the very large network setup would be interesting. It also could make sense to make the simulation environment even more realistic by studying even more complex network topologies. A second interesting topic are dynamic allocation schemes that use splittability to realize a smooth rather than an abrupt adaptation to dynamically changing traffic patterns. These are topics that we plan to investigate in future work.

References

1. P. Barford, M. Crovella. *Web Server Workload Characterization: The Search for Invariants.* Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems 1996, pp. 126–137
2. J. H. Cowie. Scalable Simulation Framework API Reference Manual, 1999. Available via *http://www.ssfnet.org*
3. M. Crovella, A. Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes.*IEEE/ACM Transactions on Networking'96.*
4. M. R. Garey and D. S Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* Freeman, 1979.
5. St. Kwek and K. Mehlhorn. Optimal search for rationals. *Information Processing Letters*, to appear, 2002.

6. P. Krysta, P. Sanders and B. Vöcking. Scheduling and Traffic Allocation for Tasks with Bounded Splittability. *Technical Report no. MPI-I-2003-1-002*. Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2003. *http://domino.mpi-sb.mpg.de/internet/reports.nsf/NumberView/2003-1-002*.
7. J. F. Kurose and K. W. Ross. *Computer Networking: a top-down approach featuring the Internet.* Addison-Wesley, 2001.
8. R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1-12, 1959.
9. C. H. Papadimitriou. Efficient search for rationals. *Information Processing Letters*, 8:1–4, 1979.
10. H. Shachnai and T. Tamir. Multiprocessor Scheduling with Machine Allotment and Parallelism Constraints. *Algorithmica*, 32(4): 651–678, 2002.
11. Scalable Simulation Framework Research Network (SSFNet).
http://www.ssfnet.org/