Distributed Adaptive Routing in Communication Networks

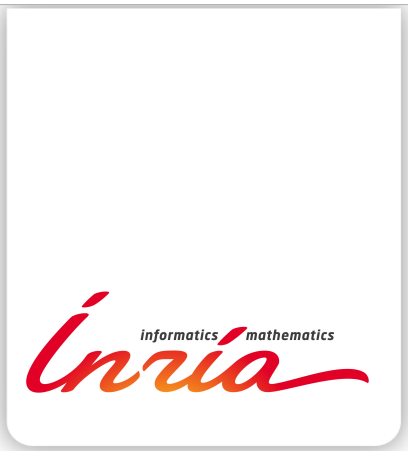
Baptiste Jonglez, Bruno Gaujal

Distributed Adaptive Routing in Communication Networks

Baptiste Jonglez, Bruno Gaujal

RESEARCH REPORT
N° 8959
October 2016
Project-Team Polaris

ISSN 0249-6399 ISRN INRIA/RR--8959--FR+ENG



Distributed Adaptive Routing in Communication Networks

Baptiste Jonglez, Bruno Gaujal

Project-Team Polaris

Research Report n° 8959 — October 2016 — 25 pages

Abstract: In this report, we present a new adaptive multi-flow routing algorithm to select end-to-end paths in packet-switched networks. This algorithm provides provable optimality guarantees in the following game theoretic sense: The network configuration converges to a configuration arbitrarily close to a pure Nash equilibrium. In this context, a Nash equilibrium is a configuration in which no flow can improve its end-to-end delay by changing its network path. This algorithm has several robustness properties making it suitable for real-life usage: it is robust to measurement errors, outdated information and clocks desynchronization. Furthermore, it is only based on local information and only takes local decisions, making it suitable for a distributed implementation. Our SDN-based proof-of-concept is built as an Openflow controller. We set up an emulation platform based on Mininet to test the behavior of our proof-of-concept implementation in several scenarios. Although real-world conditions do not conform exactly to the theoretical model, all experiments exhibit satisfying behavior, in accordance with the theoretical predictions.

Key-words: Multi-commodity routing, SDN, Nash equilibria, Adaptive routing

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Routage Distribué Adaptatif dans les Réseaux de Communication

Résumé : Dans ce rapport, nous présentons un nouvel algorithme de routage adaptatif multi-flots qui calcule des chemins bout en bout dans des réseaux à commutation de paquets. Cet algorithme a des propriétés d'optimalité au sens de la théorie des jeux: La configuration des flots dans le réseau converge vers un ε-équilibre de Nash pur. Dans ce contexte, un équilibre de Nash est une configuration dans laquelle aucun flot ne peut améliorer son délai bout en bout en changeant son chemin dans le réseau. Cet algorithme a des propriétés de robustesse qui le rende utilisable en pratique: il est robuste face aux erreurs de mesure, aux retards d'informations, et aux désynchronisations d'horloges. De plus, il n'utilise que des informations locales et ne prend que des décisions locales, ce qui le rend implémentable dans des routeurs. À partir de cet algorithme, nous avons conçu un prototype de contrôleur SDN, et fait une émulation réseau reposant sur Mininet pour tester les performances de l'algorithme sous plusieurs scenarios. Bien que les conditions du monde réel ne se conforment pas complètement au modèle théorique, nos tests montrent un comportement satisfaisant du protocole de routage, en accord avec les prédictions théoriques.

Mots-clés : Routage multi-flots, SDN, Équilibre de Nash, Routage adaptatif

1 Introduction

Communication networks are becoming increasingly multipath. Backbone networks have long been designed with multiple redundant paths, in order to increase reliability and performance. Datacenter networks need high bisection bandwidth for applications like MapReduce, which are typically obtained using multiple parallel paths in topologies such as fat-tree [3]. At the network edge, multihoming [2,30] is increasingly common, due to the reduced cost of Internet access and prevalence of mobile networks such as LTE, as a complement to fixed landline Internet access.

A common challenge is to exploit this path diversity to increase performance and/or reliability. This problem can be modeled as a *multi-commodity flow problem* [18]. Given a number of concurrent source-destination flows, the problem is to assign these flows to network paths, while respecting capacity constraints and optimizing a performance metric. Our goal is to provide a *distributed* solution to this problem, that requires neither cooperation between sites nor knowledge of the network topology and performance parameters. We restrict ourselves to the *non-splittable* case, in which a given source-destination flow uses only one path at a given time. This is necessary in practice to avoid persistent packet reordering, which would harm performance. However, a given flow is allowed to change its choice of path several times during its lifetime.

1.1 Related work

On the theoretical side, similar problems have been studied. When optimizing for a single flow with no *a priori* knowledge about the network, a multi-armed bandit modelisation [6,10] can be used for learning the shortest-delay path. However, since we consider multiple competing flows, we need a more complex modelisation based on game theory.

Regarding the multi-flow problem, convergence results for a wide class of online decision algorithms have been proven in [20], but in a different setting. The authors consider non-atomic routing games, *i.e.* each packet in a given flow independently chooses a path. This setting is hard to apply in a practical network because of packet reordering.

On the practical side, most existing approaches for exploiting multiple paths in a network are either limited to very specific settings, *e.g.* ECMP (Equal-Cost Multi-Path forwarding) for parallel paths with similar characteristics, or targeted at datacenter networks [16,32]. As such, they typically assume a network with a very organized and hierarchical topology, for instance fat-tree, or require explicit information sharing from the network [32]. We make no such assumptions.

When working with end-hosts directly, Multipath TCP [8] allows using multipath paths for a single source-destination flow, improving performance and reliability. This is the most promising approach so far, but requires extensive modifications in the operating system of both ends of the communication, which hampers large-scale deployment for now. In contrast, our proof-of-concept only requires end-hosts to include TCP timestamps or use a LEDBAT-based transport protocol, which is already widespread [31].

A noteworthy class of routing algorithms bases its routing decisions on dynamic properties of the network, including real time link load, end-to-end latency, or packet loss. This class of strategies is known as *adaptive routing*. Such solutions are attractive because they typically enable more efficient usage of network resources. In contrast, traditional load-oblivious routing algorithm may blindly over-utilize some links, leading to congestion, while other links have unused capacity. However, despite years of research, adaptive routing techniques did not see wide adoption. The main reason is the presence of potential instabilities and routing oscillations, which could make the cure worse than the disease. Indeed, early experiments with delay-based routing in the ARPANET resulted in severe stability issues under high load, rendering the

network close to unusable [19]. A possible tradeoff is to adapt routing paths at a much slower timescale. For instance, Link Weight Optimisation [17] chooses paths based on a "representative" traffic matrix, which is typically updated no more than once a day. This solves the stability issue, but it is no longer possible to adapt to real-time network conditions. More recent solutions have been proposed for adaptive routing [16, 21, 32]. Some of these solutions are based on heuristics and offer no real guarantees on their performance. Furthermore, stability issues are typically overlooked, or no convergence guarantee is provided.

1.2 Contributions

In this work, we present a novel algorithm for adaptive routing in packet-switched networks, mapping source-destination flows to paths. We claim that it provides a viable and stable solution to adapt to traffic conditions, and effectively avoids congestion. Our algorithm is based on strong theoretical grounds from game theory, while our proof-of-concept leverages SDN protocols to ease implementation. Our routing algorithm is endowed with the following desirable properties for efficient implementation:
- It is fully distributed: only local information is needed, and it requires no explicit coordination between routers;
- It is oblivious to the network topology;
- It is robust to outdated and noisy measurements;
- There are no endless oscillations;
- It does not require clock synchronization between routers, or between routers and end hosts.

We start by describing the high-level algorithm, assuming an idealized routing problem. This model assumes that the network allows source routing, i.e. end-hosts can choose to send packets along any network path to a destination. We then derive a practical algorithm that is equivalent to the first one, but readily implementable in a traditional next-hop forwarding paradigm. It operates in a more limited setting: a number of multihomed sites are connected at the edge of a core network. The algorithm then runs in the *gateway* of each site, which has multiple paths to the core network. In short, only gateways take routing decisions on behalf of flows, and the choice is limited to a choice of a next-hop router for each source-destination flow traversing the gateway.

We then describe our proof-of-concept implementation of this second algorithm, using Openflow to 1) program each router 2) collect feedback about the state of flows traversing the router. Note that our algorithm is fully distributed and does not require a central view of the network: as such, each router can be controlled by its individual SDN controller, typically collocated with the router itself.

Finally, we report and comment on experiments using Mininet [22] to evaluate the performance of our proof-of-concept in various scenarios.

2 Distributed Routing over a Network

2.1 Problem description and definitions

Let (V, E) be a communication network over a set V of nodes and a set E of bi-directional communication links, over which we consider the following *multi-commodity flow problem.*

A set $\mathcal{K}$ of *flows* of packets must be routed over the network. Each flow $k \in \mathcal{K}$ is characterized by a source-node a_k, a destination-node b_k and a nominal arrival rate of packets, λ_k. Also, each flow is affected a set $\mathcal{P}_k$ of possible paths in the network from its source to its destination, with

$|\mathcal{P}_k| = P_k$. Notice that $\mathcal{P}_k$ can either include all possible paths from a_k to b_k, or a single path, or any set in between these two extreme cases. A *configuration* is a choice of one path per flow.

Our objective is to find a configuration that minimizes a performance index. Here, we choose to minimize *end-to-end delays* of each flow. Actually, other criteria could have been chosen, such as loss rate, round trip times, or goodput, that would require a minimal adaptation of our algorithm.

This type of question has been heavily studied in the literature in many different ways [18]. Our objective here differs from most previous work: We design a *learning algorithm* that allows each flow to discover a path, such that the global configuration is a Nash equilibrium of the system. Namely, after convergence, no flow can improve its delay by changing its path.

The challenge is to consider a realistic scenario in which no flow has information about the choices of the others or is even aware of the presence of other flows. The only information that a flow can get from the network is an estimation of the end-to-end delay of its packets sent over its current chosen path. To make things even more realistic, we assume that the delay measurements can be perturbed by random noise.

The difficulty also comes from the interdependence between the flows: when one flow decides to send its packets on a new path, this may alter the delays for the other flows because of resources sharing between the paths (either links or routers).

In this context we design a distributed algorithm that allows each flow k to eventually choose a path in $\mathcal{P}_k$ such that all alternative paths would offer a larger delay. This goal is achieved by using optimization methods coming from game theory.

An illustrative example is given in Figure 1 with three flows. Flow 1 can be seen as the "main" flow while the other two flows are cross traffic that may alter the performance of the main flow.

We start from the configuration 1.(b), where all flows share some links, hence inducing large delays. The goal is to achieve one of configuration 1.(c) or 1.(d), by letting all flows explore simultaneously their paths and achieving such a coordination in a fully distributed way, in the sense that no flow has information about the presence of other flows and only has local information about the network.

2.2 OPS Algorithm

Here is a description of the algorithm that each flow k runs independently. It is based on a mirror-descent algorithm for general potential games, presented in [15]. In the following, the index k may be dropped when no confusion is possible, but one must keep in mind that all flows execute the same code, not necessarily synchronously.

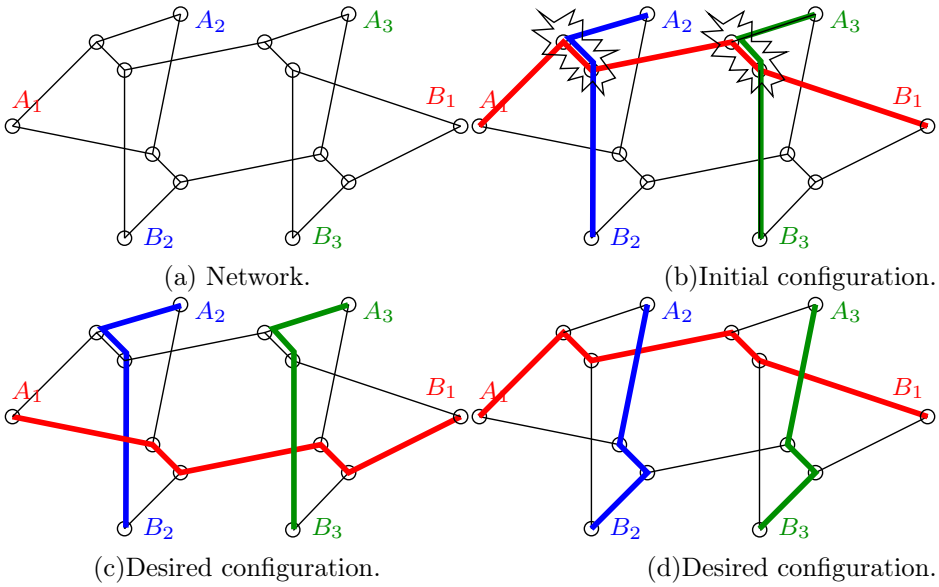
For one flow, say k, we call $d_k(p_1, \cdots, p_k, \cdots, p_K)$, the end-to-end *average delay* for packets of flow k under the configuration where flow 1 uses path p_1 among its possible paths, flow 2 uses path p_2, and so forth.

The algorithm executed for flow k is probabilistic and maintains two vectors, both of size P_k (denoted P to ease notations).

The *probabilistic choice* vector $\mathbf{q} = (q_1 \dots q_P)$ gives the probability to choose each path p.

The *score vector* $\mathbf{Y} = (Y_1 \dots Y_P)$ maintains a (negative) score for each path, to be optimized, where Y_p depends on the average delay for packets of flow k on path p.

The main loop of the OPS algorithm (Algorithm 1) is as follows. Each time a local timer ticks (further details on this are provided in Section 3), a path p is chosen according to the probability distribution $\mathbf{q}$, and packets are sent along p. The delay of packets over this path is measured. The score Y_p is updated according to a discounted sum and in turn, the probability vector is modified for the next path selection using a logit distribution. This repeats forever, or



(a) Network. (b)Initial configuration.

(c)Desired configuration. (d)Desired configuration.

Figure 1: (a) displays a network with 3 flows $(A_1, B_1), (A_2, B_2), (A_3, B_3)$. (b) shows the initial configuration that suffers from congestion in two links (marked by star shapes). The final configurations (c) and (d) have no congestion.

until a stable path has been reached for all flows, *i.e.* $\mathbf{q}$ becomes a degenerate probability vector (all coordinates are equal to zero except one) for all flows.

Algorithm 1: Optimal Path Selection (OPS) Algorithm for flow k.

Initialize:
$n \leftarrow 0$; $\mathbf{q} \leftarrow (\frac{1}{P}, \dots, \frac{1}{P})$; $\mathbf{Y} \leftarrow (0, 0, \dots, 0)$;
repeat
  When local timer ticks for the nth time;
  $n \leftarrow n + 1$;
  select a new path p w.r.t. probabilities $\mathbf{q}$;
  Use path p and measure delay D;
  Update score of path p: $Y_p \leftarrow \left(Y_p - \gamma_n (D + \tau Y_p)/q_p \right) \vee \beta_n$;
  foreach *path* $s \in \mathcal{P}_k$ **do**
    update probability: $q_s \leftarrow \frac{\exp(Y_s)}{\sum_\ell \exp(Y_\ell)}$;
until *end of time*;

The OPS algorithm uses 3 parameters: $\tau > 0$ is a discounting factor over past scores. The bounding sequence β_n (in the algorithm, $\vee$ denotes the maximum operator) is such that $\beta_n \to -\infty$ and $|\beta_n| \leqslant C_1 n + C_2$ for some constants C_1 and C_2. The decreasing sequence of discretization steps γ_n is in L_2 ($\sum_n \gamma_n^2$ converges), but not in L_1, ($\sum_n \gamma_n$ diverges). Typically, $\gamma_n = 1/n^\alpha$ with $1/2 < \alpha \leqslant 1$ works.

Discussion on how to tune these parameters is postponed to Section 3.6.

2.3 Convergence Properties

Assumption 1. *The measurement $D(t)$ of the delay is not biased. More precisely, the measurement done by flow k, over path p_k at physical time t is such that $D(t) = d_k(p_1(t) \dots p_k(t) \dots p_K(t)) + \xi(t)$, where the noise $\xi(t)$ is a non-biased random variable, conditionally to the past, $\mathcal{F}_t$: $\mathbb{E}(\xi(t)|\mathcal{F}_t) = 0$, and has a finite second moment: $\mathbb{E}(\xi(t)^2|\mathcal{F}_t) < \infty$.*

Recall that d_k is the average end-to-end delay experienced by flow k in a given configuration. This is a rather mild assumption, because the errors on the measurements do not need to be bounded, nor does it have to be independent of the current configuration of the traffic, for example a large load over one link may induce a noise with large variance on that link.

Assumption 2. *All the flows have the same arrival rate.*

This assumption is technical and deserves some comments. It is needed to make sure that the underlying game has a potential [25]. This is an essential ingredient to prove convergence (see Appendix A).

However, it is not realistic: In general, all flows do not have the same arrival rate. In that case, one can still use the OPS algorithm. It can be shown (not reported here) that if OPS converges, it finds a Nash equilibrium. However, convergence is not guaranteed in general. An alternative, when all rates are commensurate, is to split flows into subflows, all with the same rate. In that case the subflows from the same flow may end up using different paths from source to destination. This solution is not ideal because it implies re-ordering of packets at the destination. However, if the arrival rate of the sub-flows is small enough, the delays on the different paths will be very similar after convergence, so that re-ordering will be minimal. The similarity of delays on different paths comes from the fact that, with small arrival rates, Nash equilibria approach Wardrop Equilibria, under which all the packets of a flow experience the same delay.

Theorem 1 (Convergence to equilibrium)**.**
Under assumptions 1 and 2, for all $\epsilon > 0$, there exists $\tau > 0$ such that under discounting factor τ, Algorithm 1 converges for all flows to an ϵ-optimal configuration, in the following sense:
For each flow k, the probability vector $\mathbf{q}$ converges almost surely to a near degenerate probability: q_p becomes smaller than ϵ for all $p \in \mathcal{P}_k$ except for one path, say p_k^*, for which it grows larger than $1 - \epsilon$.
Furthermore, under configuration $(p_1^*, \dots, p_K^*)$, no flow can unilaterally reduce its delay: for all k and $\forall p \in \mathcal{P}_k$, $d_k(p_1^*, \dots, p, \dots, p_K^*) \geqslant d_k(p_1^*, \dots, p_k^*, \dots, p_K^*)$.

The proof of Theorem 1, given in Appendix A, also shows the following additional properties of the OPS Algorithm.

The algorithm is multiflow. The algorithm is distributed and multiflow in the sense that all the flows execute the algorithm in parallel and they all converge to a stable configuration.

The algorithm is incrementally deployable. Not all flows in the network need to use the OPS algorithm. If only a subset of the flows use the OPS algorithm, while all the other flows are forwarded using static routing tables, the convergence property still holds. The flows being forwarded statically are seen as part of the (random) environment over which optimization is done.

OPS is only based on online local information. The only information used by the flows is the current delay on their current path. They do not know the topology of the network, the capacity of the network, the number of concurrent flows or the paths taken by the other flows. The discovery of the best path for each flow is done "in the dark", without any exchange of information or any kind of collaboration between the flows or with the network.

Asynchronous updates. The algorithm is truly asynchronous: convergence will occur even if the different flows update their choices using different timers, that can have arbitrary initial offsets as well as arbitrary different speeds. Of course, the speed of convergence may be slow if flows have very different update frequencies.

Robustness to errors. As explicitly specified in Assumption 1, convergence to the equilibrium is robust to measurement errors. Actually, this is very useful in our context, not only because measurements do not have an infinite precision, but mainly because the measurement of the delay over one path is done by monitoring one or several packets (see Section 3.2 for a detailed description on how this can be done) and by computing the empirical mean. This empirical mean equals the average delay up to some additive random term. This term satisfies Assumption 1 as soon as the system is stationary.

Robustness to outdated measurements. The estimation of the delay D on path p is based on monitoring the delay of several packets taking this path. When the score Y_p is computed and a new path is selected, these measurements are outdated because other flows may have already changed their paths. The theorem shows that this phenomenon does not jeopardize the convergence properties of the algorithm.

Convergence to a non-randomized choice. Theorem 1 says that there are no endless oscillations. The flows explore their different options in a transient phase of the algorithm. This creates some oscillations in the traffic. However, convergence to a stable point starts to settle in after some time and complete stabilization to the choice of a single path for each flow is always achieved.

2.4 Relaxing convergence

The guarantee of almost sure convergence requires that the step-size sequence γ_n vanishes to 0. One can relax this vanishing condition and show that for constant step sizes ($\gamma_n = \gamma, \forall n \in \mathbb{N}$), convergence occurs in probability instead of almost surely (see [9]). Namely, the distribution of $\mathbf{q}$ concentrates to near degenerate probability distributions when γ and τ go to 0.

2.5 Speed of convergence

The proof of the theorem given in Appendix A is based on the construction of a stochastic approximation of a differential equation and does not provide any information on the speed of convergence of the algorithm to a Nash equilibrium. The speed of convergence of similar algorithms has been proved to be of order $1/n$ [24] in a centralized context (a single player) and with strictly convex objective functions. Neither conditions are true here (many players with arbitrary delay functions). To our knowledge, no theoretical result exists today to bound the speed of convergence for the type of algorithms used here, so it must be evaluated experimentally.

From a practical point of view, several factors can affect the convergence of the OPS algorithm.

- Size of the network: The algorithm being based on the measurement of end-to-end delays, the size of the network should not play a decisive role in the convergence speed. The effects of the network size are indirect: a larger network means a larger number of possible paths for each flow, and more potential for delay measurements to be outdated.

- Number of players: Classical algorithms let players play one at a time, while our OPS algorithm makes them play simultaneously. This speeds up convergence, especially with a large number of players. Several simulations of OPS done in a different context (not reported here) suggest that the speed of convergence does not depend crucially on the number of players.

- Number of possible paths per player: In contrast with the number of players, the convergence time is expected to increase drastically when the number of choices per flow increases. Each flow needs to probe each path a sufficient number of times before it can assess its performance.

- Precision of the measurements: It should be clear that when the measurements of the end-to-end delays suffer from a large variance, then convergence slows down. We believe that this is the most important parameter that influences the convergence time. This belief is reinforced by the experiments reported in Figure 5.

2.6 Price of Anarchy

In general, a Nash equilibrium (NE) does not provide any guarantee on its global performance. In the worst case, the sum of the delays of all flows under a Nash equilibrium can be arbitrarily far from an optimal configuration. However, there are more favorable situations in which Nash equilibria can provide performance guarantees.

Here, the social cost of any path configuration $\mathbf{p} = (p_1, \ldots, p_K)$ is the total sum of the average delays that can be decomposed over all links of the network, $c(\mathbf{p}) := \sum_{e \in E} N_e(\mathbf{p}) \cdot \delta_e(N_e(\mathbf{p}))$, where $\delta_e(\ell)$ is the average delay on link e under load ℓ and $N_e(\mathbf{p})$ is the number of flows using link e under configuration $\mathbf{p}$.

A first argument in favor of NE is that social optima that are not NE are unstable configurations, because some flows can gain by changing their paths, and will certainly do so if they are not constrained by some kind of central authority.

Moreover, in some cases, NE exhibit good global performance. This has been deeply investigated in the literature: we simply provide a quick review here. In general, the *price of anarchy* (ratio between the social cost of the worse Nash equilibrium over the cost of the social optimum) is upper-bounded by $\Theta(\log n)$ [12] where n is the number of players.

Furthermore, if the delays of all links are (λ, μ)-smooth (see the definition in [12]), then the price of anarchy is bounded by a constant, namely $\frac{\lambda}{1-\mu}$. As a special case, this bound becomes $5/2$ when the delay on each link of the network is affine w.r.t. its load (see [12]). Additionally, when the delays are arbitrary increasing functions of the load, the social cost of any NE is smaller than the optimal social cost of the same congestion game where rates are doubled [28].

There exists cases where NE are local social optima. This is for example the case when the potential function (minimized by NE), $\phi(\mathbf{p}) = \sum_{e \in E} \sum_{i=0}^{N_e(\mathbf{p})} \delta_e(i)$ approaches the social cost $c(\mathbf{p})$. This can happen when the delays do not depend on the load of links ($\mathrm{M/M/\infty}$ models for the delay on all links). However, this may not be the most interesting case, because then the flows will not interfere with each other at all.

Finally, several recent publications question the worst case bounds found in [26] and show that in real life cases with a large number of players [13] or when the arrival rate grows [14], the price of anarchy approaches one.

3 Implementation as a SDN controller

We now focus on the more practical aspects of our model, and discuss how Algorithm 1 can be implemented in a real network. We then describe a more practical version of our algorithm, designed as a SDN controller. We have successfully implemented this SDN-based algorithm, and we use this implementation to run experiments in Section 4.

3.1 Running our algorithm in gateways

Algorithm 1 assumes that flows, or equivalently end-hosts, can choose the full path to their destination (line 6 of the algorithm). This paradigm is known as *source routing*, but is not widely deployed in the public Internet: source routing was deprecated in IP because of security issues [RFC 5095]. Thus, we first make assumptions on the structure of the network, and we then adapt Algorithm 1 so that routing decisions are taken by routers on behalf of end-hosts.

First, we assume that the network can be decomposed into a *core* network and several *edge* networks. Each edge network connects to the rest of the network through a single router, which we call a *gateway*. Gateways may connect directly to each other, or through core routers. Figure 2 shows an example with three edge networks, each of which contains a gateway and one or more hosts.

Then, to ease implementation, we assume our OPS algorithm is only run by gateways to select a path among those offered by the core network. This way, core routers do not need to keep state for each source-destination flow. Core routers simply use their regular routing protocols, for instance BGP or OSPF. They can also use ECMP (Equal-Cost Multiple Paths) with a hashing mechanism on the packet source and destination, so that a given source-destination flow in the core network is forwarded along a unique path. This is necessary to obtain consistent delay measurements during the lifetime of a flow.

Limiting our algorithm to only run in gateways means that a flow cannot explore all possible paths to its destination: the set of possible paths $\mathcal{P}_k$ defined in Section 2 for flow k is limited to the next-hop routers of the gateway. Still, even with two or three paths to choose from, this provides enough path diversity to make our algorithm interesting to run.

The decomposition of the network into gateways and core routers is illustrated in Figure 2.

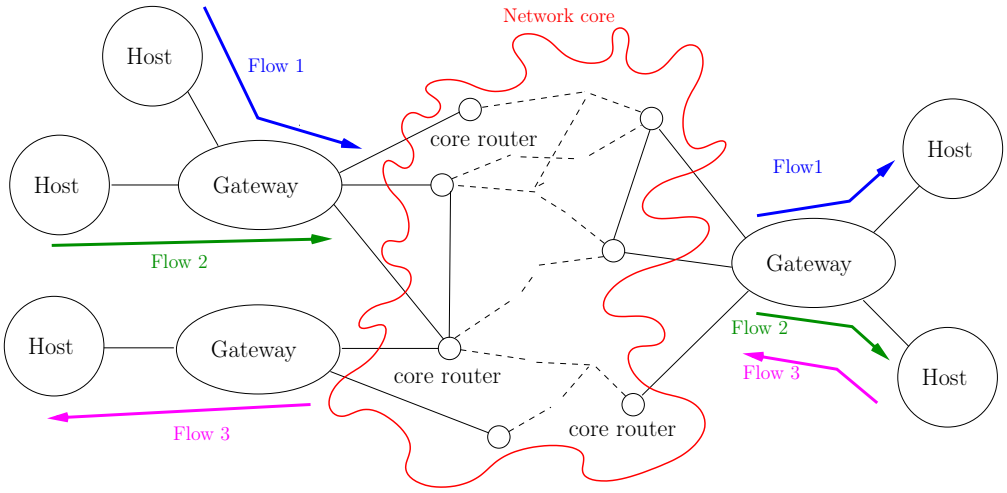


Figure 2: Illustration of the network decomposition into hosts, gateways and routers. An example of 3 flows between hosts is also displayed.

3.2 Exploiting one-way delay

Algorithm 1 uses a generic notion of "delay" as objective function (line 7 of Algorithm 1). We focus on end-to-end *one-way delay* instead of the more classical RTT. There are several reasons to this choice:

1) For latency-critical applications such as Voice-over-IP, the relevant metric is often the transmission delay on the forward path.
2) A router has no control over the reverse path since routing decisions only affect the *forward* path of a flow towards its destination. For instance, the LEDBAT congestion control algorithm [29] also exploits one-way delay for the same reason: it allows congestion detection and bufferbloat avoidance on the forward path, while ignoring the effect of the reverse path.

To passively determine the one-way delay of forwarded packets, we use *timestamps* that some transport protocol include in their headers: we focus on TCP with the Timestamp Option and μTP, an implementation of LEDBAT [29] over UDP. To estimate one-way delay from these timestamps, we use the same method as [11] to extract one-way delay samples from TCP and μTP/UDP packets, and then normalize the measurements by expressing them in milliseconds.

In our case, the gateway is the "observation point" at which the delay is observed. A gateway is interested in the one-way delay from A to B, where A is a host in the local edge network, and B is a remote host. The *observed one-way delay* is measured for every packet i from A to B, and we call it $\delta_{\text{observed}}(i)$. It is the only quantity we can measure directly, but it can be theoretically decomposed in this fashion:

$$\delta_{\text{observed}}(i) = \delta_{\text{propagation}}(p_i) + \delta_{\text{queuing}}(i) + \text{off}_{A,B}$$

where p_i is the path taken by packet i in the network. Here, we consider three contributions to the observed one-way delay:

1. the clock offset of B relatively to A, $\text{off}_{A,B}$. This quantity is independent of the path taken by a packet, and we further assume that it does not change over time.

2. a delay contribution that depends only on the path taken by a packet, $\delta_{\text{propagation}}(p_i)$. This is the one-way propagation delay defined by physical laws (*e.g.* the speed of light in a fiber) and by the constant switching time of switches and routers on the path. By definition, this quantity is the same for all packets forwarded on the same path.

3. the remaining contribution, which is specific to each packet. We call this last contribution $\delta_{\text{queuing}}(i)$, because it is mostly made of queuing delays at switch and router interfaces along the path. This contribution actually also includes serialization delay, although it is negligible in high speed networks.

To eliminate the clock offset and get positive estimates of the delays (needed to guarantee proper convergence of the algorithm), we compute the minimum of the observed delay measurements, similarly to LEDBAT. In our case, this minimum is taken over all paths selected by the algorithm. Given a measurement of the observed delay $\delta_{\text{observed}}(i)$ on the ith packet, with p_i being the path selected for this packet, the *relative delay* we consider for our algorithm is the following:

$$\begin{aligned}
\Delta(i) &:= \delta_{\text{observed}}(i) - \min_{0 \leqslant l \leqslant i} \delta_{\text{observed}}(l) && (1)\\
&= \delta_{\text{observed}}(i) - \min_{0 \leqslant l \leqslant i} \left[\delta_{\text{propagation}}(p_l) + \delta_{\text{queuing}}(l) + \text{off}_{A,B}\right] && (2)\\
&= \delta_{\text{observed}}(i) - \left[\text{off}_{A,B} + \min_{\text{path } p} \delta_{\text{propagation}}(p)\right] && (3)\\
&= \delta_{\text{propagation}}(p_i) + \delta_{\text{queuing}}(i) - \min_{\text{path } p} \delta_{\text{propagation}}(p). && (4)
\end{aligned}$$

Going from (2) to (3) requires that i is large enough to achieve a minimum queuing delay of zero, and to ensure all paths have been visited. In practice, our algorithm starts with a uniform

probability to select each path, so most paths are expected to be probed before the algorithm starts to converge.

This way, we eliminate the clock offset and obtain a positive estimate of the one-way-delay on the current path p_i, relatively to the path with the lowest propagation delay. It can be shown that this additive shift does not affect the behavior of the algorithm, given that $\Delta(i)$ remains non-negative.

3.3 Update policy

Algorithm 1 (line 4) uses a timer for its updates, denoted n. This becomes a periodic update in the implementation: for each flow forwarded by a gateway, the gateway takes a routing decision every T packets of this flow. Among these T packets, the gateway only uses the one-way delay of the last S of them and compute their empirical mean. Using the last packets of the period helps to ensure that the transient state associated with the last change of path has disappeared, and that measured delays reflect the steady-state performance of the flow.

Recall that $\Delta(i)$ is the *relative delay* of packet i, defined in Section 3.2. Using the above notations, the delay D used in line 7 of Algorithm 1 is computed as:

$$D := \frac{\Delta(nT - S + 1) + \cdots + \Delta(nT)}{S}.$$

This choice has several advantages:

- Large flows (*i..e.* with large arrival rates) are more likely to affect the delays of other flows. Using this construction of D makes sure that they are updated more frequently.
- This construction also provides some control on the random error ξ (defined in Assumption 1): A large T ensures that the expectation of ξ is close to 0 while a large S reduces its variance.

3.4 SDN-based implementation

We are now ready to adapt Algorithm 1 so that it is more practical for real networks. Below is a description of the main ideas of our implementation, while Section 3.5 contains a more formal description of the practical algorithm we implemented. The code of our actual implementation is available online[1].

Our proof-of-concept implementation takes the form of an Openflow controller, using the Ryu [1] library. A gateway is made of both an Openflow switch and a dedicated controller, but we use the term "gateway" to refer to the Openflow switch only.

The forwarding table of a gateway is programmed and constantly updated by its controller. Furthermore, a gateway sends a copy of packet headers back to its controller, for delay computation. The controller is configured beforehand with a static multipath routing table. That is, for each IP prefix, the routing table lists all possible next-hop routers that can be used to reach the destination.

When a gateway receives the first packet of a new flow, it forwards it to the controller, to receive instructions for subsequent packets of the same flow. The controller takes an initial routing decision for this flow, according to its routing table. This decision is a choice of a next-hop router through which packets will be forwarded. A corresponding forwarding rule is then installed into the gateway.

[1] `https://gforge.inria.fr/projects/derouted`

To measure the performance of each decision, the controller also installs a rule to receive a copy of all packet headers. Upon receiving a header, the controller extracts timestamps, computes the end-to-end one-way delay of the flow, and updates the score of the current choice. For each active flow independently, the controller periodically decides to select a new next-hop router, based on the scores. Each time the controller changes its decision, new forwarding rules are installed in the gateway.

The main reason to use a SDN framework is ease of implementation. Indeed, Openflow abstracts away the communication with forwarding elements, allowing to easily install forwarding rules and receive packet headers for performance measurements. For our experiments, we used Openvswitch on Linux, but any Openflow-compatible switch could have been used instead.

An important architectural decision was to exploit the decentralized nature of our algorithm, by ignoring the centralized management features of Openflow. Here, each gateway is controlled by a separate Openflow controller. Thus, we retain the scalability and resilience properties of decentralized routing.

Another key point is that we use Openflow switches as pure layer-3 devices. Openflow 1.3 is perfectly capable of programming a switch to act as a layer-3 router, by decreasing the TTL and modifying the MAC addresses of packets. Working at layer 2 would not make much sense, because we fundamentally solve a routing problem. Besides, it is difficult to have control on flooding when working at layer 2 in complex topologies with redundant paths.

3.5 Practical algorithm

Algorithm 2 is a version of Algorithm 1 that is much closer to the actual implementation. This algorithm is run independently for each gateway, where a flow is identified by its source and destination addresses. That is, the controller of each gateway executes this algorithm independently, for all flows currently forwarded by the gateway.

3.6 Parameters

Both Algorithm 1 and Algorithm 2 use a number of parameters. We discuss how to choose appropriate values for these parameters, and the default value we use in our implementation. These default values are then used in the experiments we present in Section 4.

Step size of score updates γ (Section 2.2) We choose a constant value of γ, instead of a decreasing sequence γ_n. This implies convergence only in a weaker sense (in probability instead of almost surely), as discussed at the end of Section 2.3. However, a constant value of γ is interesting in practice, because it reduces convergence time: in the original algorithm, when γ_n goes to zero, convergence to the equilibrium becomes slower and slower as n increases. In short, we trade a bit of theoretical guarantees for a better convergence time.

The numerical value of γ should be small compared to typical measured delays, since it is the discretization step of the differential equation driving the dynamics of the system (see Appendix A). In our implementation, we have chosen $\gamma = 0.01$, which satisfies the above constraint since we use delays expressed in milliseconds.

Discount parameter τ (Section 2.2) A smaller value means that the algorithm will converge closer to a Nash Equilibrium, but convergence will be slower. We choose $\tau = 0.1$. This allows fairly quick convergence, at the expense of a little bit of accuracy.

Algorithm 2: OPS Algorithm for the SDN controller

```
1  Initialisation
2      Populate routing table with static entries (mapping IP prefixes to lists of next-hop
       routers)
3      Install a table-miss Openflow rule in switch: priority=0, match=*,
       action=CONTROLLER
4  When packet pkt arrives at the controller
5      if pkt belongs to an unknown flow then
6          flow1 ← create_flow (pkt.src, pkt.dest)
7          take_routing_decision (flow1)
8          flow2 ← create_flow (pkt.dest, pkt.src)
9          take_routing_decision (flow2)
10     else
11         Find flow f such that f.src = pkt.dest and f.dest = pkt.src
12         Δ ← compute_onewaydelay (f, pkt)
13         Store Δ in a queue for later use
14         f.packet_count += 1
15         if f.packet_count ⩾ T then
16             D ← average of the last S values of delay Δ for flow f
17             update_score (f.current_nexthop, D)
18             take_routing_decision (f)
19             f.packet_count ← 0
   // Parameters S and T are presented in Section 3.3
   // Functions in boldface are detailed in Algorithm 3
```

Bounding sequence β_n (Section 2.2) This sequence is essentially useful for the proof and does not play an important role in practice. It is skipped in the implementation.

Update interval T (Section 3.3) By default, gateways update their choice every $T = 500$ packets for each flow. Assuming a flow with a throughput in the range of a few Mbit/s, this means our algorithm performs an update every few seconds at most. Note that an update of the algorithm does not necessarily mean that a new path is selected: the new choice can be identical to the previous choice. Thus, the rate of *actual* routing changes will be much lower than this, especially since the algorithm is expected to converge to a deterministic choice.

A few seconds between choices is a reasonable value: the network has time to adapt before measuring the performance of the new configuration. Indeed, just after the change, the queues of routers in the network might exhibit a transient behavior, which is not adequate to measure average delays. Additionally, we measure delays using packets on the reverse path. If the end-to-end RTT of a flow was higher than our update interval, then each decision would be based on outdated information. The algorithm is designed to be robust to such outdated information, but we still want to avoid it, as it may slow down convergence.

For flows with a larger throughput, $T = 500$ packets would be too low, as it would mean very frequent changes of path. A possibility would be to specify an update interval in seconds, independently of the throughput of the flow. However, this might cause fairness issues between flows with different throughput. Since we explicitly assume that all flows have the same throughput to be able to prove convergence, we have not yet explored this area.

Algorithm 3: Functions used in Algorithm 2

create_flow(src_ip, dest_ip)
    // provides multiple next-hop routers
    f.nexthops ← Lookup RIB for longest prefix match on dest_ip
    f.current_nexthop ← None
    // minimum one-way delay across all possible next-hop-router choices
    f.min_delay ← MAX_INT
    Add flow f to controller flow table
    return f
compute_onewaydelay(**flow** f, **packet** pkt)
    // see Section 3.2
    observed_delay ← compute one-way delay from timestamps in pkt header
    f.min_delay ← min(f.min_delay, observed_delay)
    return observed_delay - f.min_delay
update_score(**nexthop** nh, **delay** D)
    $\mathsf{nh.score} \leftarrow \mathsf{nh.score} - \gamma \frac{D + \tau \cdot \mathsf{nh.score}}{\mathsf{nh.proba}}$
take_routing_decision(**flow** f)
    $\mathsf{nh.proba} \leftarrow \frac{\exp(\mathsf{nh.score})}{\sum_{\mathsf{nh'}} \exp(\mathsf{nh'.score})}$ for each nexthop nh
    new_nexthop ← pick nh at random in f.nexthops with proba nh.proba
    f.current_nexthop ← new_nexthop
    Add Openflow rule:
    match: src=f.src,dst=f.dest, action: decrement TTL, update MAC addresses, forward to nh.port and send a copy to controller

Number of packets to use for the empirical mean of delays S (Section 3.3) We selected $S = 5$ packets, since we observe experimentally that this already reduces the variance of delay measurements. A larger value of S means a larger risk to be exposed to transient delay values.

Summary of parameters The various parameters used in the implementation and their default value are summarized here:

γ	τ	T	S
0.01	0.1	500 packets	5 packets

4 Experimental Evaluation

We next evaluate our OPS algorithm in a simple network, to make sure it discovers optimal paths in practice. We are also interested in the time it takes to reach a solution.

The experiments were performed using emulation on Linux systems. This means that we were able to run our SDN implementation unmodified: we thus expect our results to be quite close to reality. Using emulation also means that we were unable to perform a large number of experiments or use large-scale topologies. Indeed, setting up and then running each experiment in a reproducible manner requires writing the configuration of each router and controller.

We used Mininet [22] to emulate network topologies, running on a single server. Although realistic in a functional sense, Mininet is known to exhibit unrealistic performance characteristics [22]. To prevent this effect, we limit the capacity of the links to very low values using netem (in the order of 10 Mbit/s). Furthermore, we run our Mininet network on fast and modern server hardware, which should alleviate performance concerns given the low throughput involved in the experiments.

4.1 Experimental setup

To evaluate our algorithm, we consider a network satisfying the conditions laid out in Section 3.1. This network is simple enough to understand how the algorithm behaves, but complex enough so that the solution found by the algorithm is non-trivial. The network is shown in Figure 3. A set of *gateways* G_1 to G_4 are connected to each other, and to a central router R that models the *core* of the network. A set of *hosts*, each attached to a gateway, send long-lived UDP flows to each other. Five UDP flows are considered, numbered from 1 to 5: from Host 1 to Host 3, from Host 4 to Host 1, from Host 4 to Host 2, from Host 4 to Host 3, and finally from Host 2 to Host 3.

For this evaluation, all flows have a constant throughput equal to λ, and have an infinite duration. It can be seen as a model of a particular application usage, for instance a network used to transport multiple real-time video streams.

Note that our OPS algorithm only runs in the four gateways. The central router R simply forwards packets using a static routing table, designed to minimize the number of hops to the destination. In this case, all gateways are one hop away from R, so the routing table trivially forwards packet directed to host i through gateway G_i.

Additionally, each gateway uses static routes for packets coming from other gateways since the OPS algorithm is only run for packets from its own hosts (see Section 3.1). The next-hop router of these static routes is the destination gateway if it is a neighbor, and R otherwise.

The capacity of each link is detailed in the Figure. To ease explanations, we introduce the *load* ρ as $\rho = \frac{\lambda}{4000 \text{ Kbit/s}}$. A load of 1 means that a single flow would occupy the full capacity of a 4 Mbit/s link.

As long as $\rho < 1$, there exists at least one stable configuration of the network, *i.e.* a choice of path for each flow that satisfies capacity constraints on all links. Additionally, for $\frac{2}{3} < \rho < 1$, there are only two possible stable configurations, described in Figure 4.

Since we have four gateways, we run four instances of our Ryu-based Openflow controller, each controlling a different emulated gateway. Once Mininet and the controllers are setup, we then use `udpmt` (part of the `ipmt` toolbox [27]) to generate UDP packets for each flow. We modified `udpmt` so that it sends packets with a μTP header, which includes a timestamp of the date of emission. The destination host for each flow runs `udptarget`, which we modified to behave similarly to a μTP implementation: it replies back with small UDP packets containing μTP timestamps.

This setup has been run a large number of times, to reduce variability. Each experiment lasts for 2400 seconds, because we found it was sufficient to allow most executions to converge. To parallelize the execution, we used Grid'5000 [7] as an IaaS platform. We reserved several identical physical machines from a cluster, deployed a Debian Jessie image with all necessary software, and ran the same experiments independently on each machine. The machines are Bullx Blade B500 servers, with a 8-core Intel Xeon E5520 CPU and 24 GiB of RAM. Depending on the experiments, we used between 5 and 45 machines in parallel.

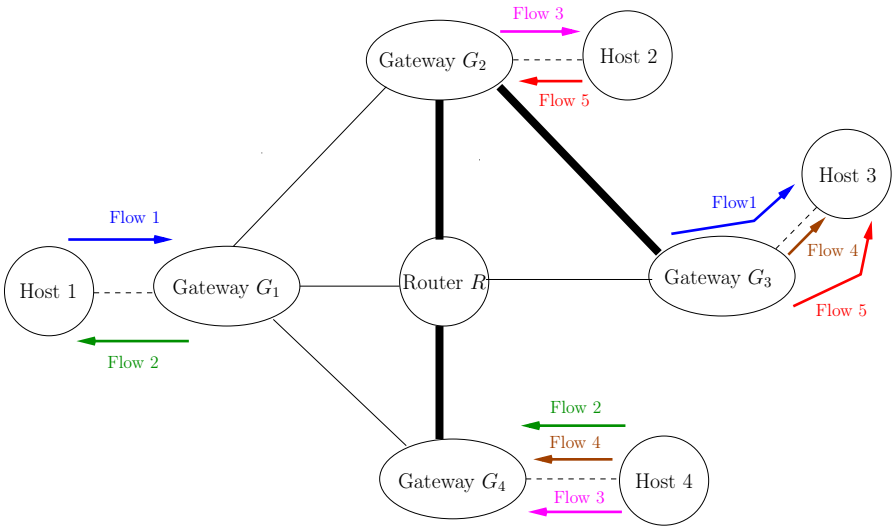


Figure 3: Network used in the following experiments. Thin lines represent links with capacity of 4 Mbit/s, while thick lines represent links with a capacity of 8 Mbit/s. All links have a transmission delay equal to 5 ms. Dashed lines between hosts and their gateway are links with unrestricted capacity. Five flows using the network are represented. Each flow consists in UDP packets with a constant throughput λ, with λ varying from 2000 to 3900 Kbit/s in the experiments.

Flow	Equilibrium 1	Equilibrium 2
Flow 1	$G_1 \to G_2 \to G_3$	$G_1 \to G_2 \to G_3$
Flow 2	$G_4 \to G_1$	$G_4 \to R \to G_1$
Flow 3	$G_4 \to R \to G_2$	$G_4 \to R \to G_2$
Flow 4	$G_4 \to R \to G_3$	$G_4 \to G_1 \to R \to G_3$
Flow 5	$G_2 \to G_3$	$G_2 \to G_3$

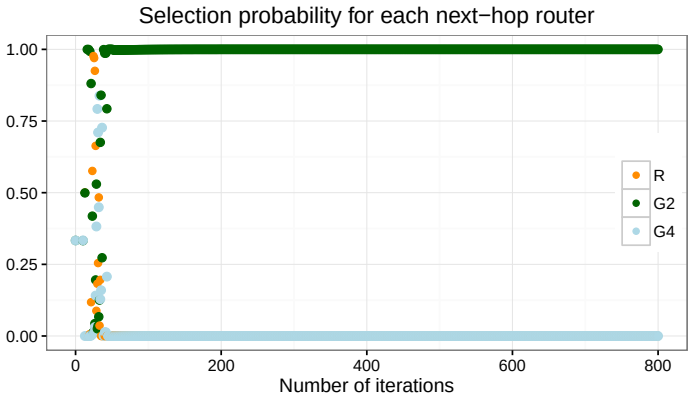
Figure 4: Stable configurations for the network of Figure 3, when the load $\rho = \frac{\lambda}{4000 \text{ Kbit/s}}$ satisfies $\frac{2}{3} \leqslant \rho < 1$.

4.2 Discussion of the results

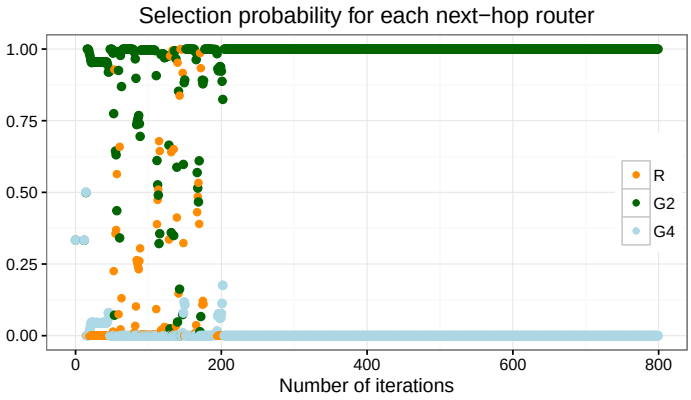
The experiment was carried under several values of load ρ ($\rho = \frac{\lambda}{4 \text{ Mbit/s}}$). In the experiments, the load varies from 0.75 to 0.975, to exercise the algorithm under moderate and heavy loads.

Figure 5 shows the evolution of Flow 1 under different loads. Gateway G_1 has three possible next-hop routers: G_2, R, or G_4. We display the probability, over time, that G_1 selects each of the three next-hop routers. The other flows are also being forwarded concurrently, but this is not displayed on the figure.

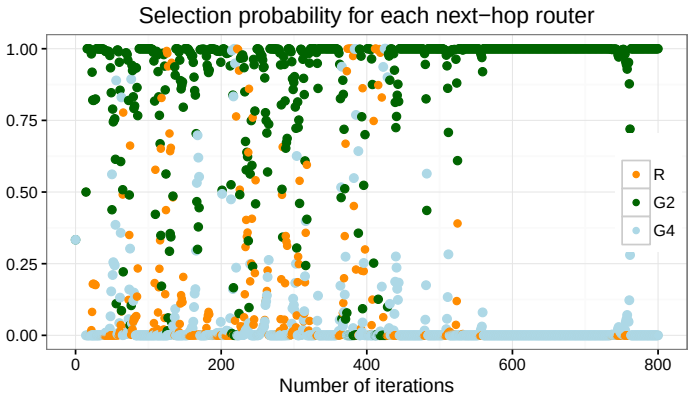
G_2 gets selected most of the time, after a transient period. This choice is consistent with both optimal configurations for the network. We also note that, when the load increases, the transient period becomes longer. Looking at Figure 5c, the convergence of Flow 1 is not obvious when the load is high. In fact, Flow 1 has converged to an "almost pure choice" as predicted by Theorem 1. This means that the flow will spend most of its time on the optimal choice, but can explore other choices from time to time. This effect is more visible when the load is high: in Figure 5c, G_2 is selected most of the time, but the flow also explores G_4 and R from time to time. Taking



(a) Load equal to 0.75



(b) Load equal to 0.875



(c) Load equal to 0.925

Figure 5: Probability of selecting each next-hop router over time, for Flow 1 (see Figure 3). Each figure corresponds to an increasing value of load for all flows. Here, $T = 500$ packets and $S = 5$ packets.

into account these observations, we have designed the following convergence criterion for a flow:

A flow has converged to path p at time t if the average probability of choosing p for the last W updates was at least Θ.

Here, W is the size of a moving window counted as a number of updates of our algorithm (recall from Section 3.3 that a gateway periodically updates the path of its flows, every T packets).

The *global convergence criterion* for a network with multiple flows is satisfied when *all* flows satisfy the convergence criterion at time t, for the same parameters W and Θ.

Using this convergence criterion over the previous experiments with $W = 50$ and $\Theta = 80\%$, we have computed the global convergence time of the algorithm under different loads. Note that the convergence time cannot be lower than $W = 50$ iterations. The results are depicted in Figure 6, where the convergence time is expressed as a number of iterations of our algorithm. For a moderate load, the global convergence time is close to the minimum of $W = 50$ iterations. When the load approaches 1, we expect that the global convergence time goes to infinity, because the variance of the delays goes to infinity.

Note that we intentionally exercise our algorithm in unfavorable conditions. For a low or moderate load, which is the usual case for over-provisioned operator networks, the global convergence time is very close to the minimal convergence time.

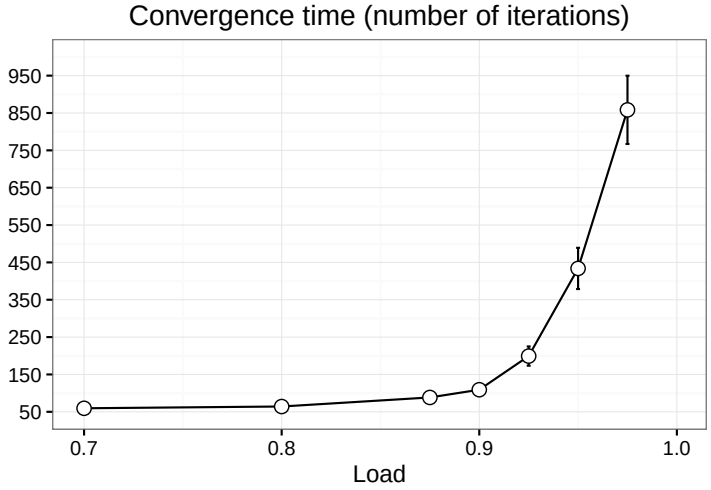


Figure 6: Average global convergence time of the algorithm as a function of the load, with 95 % confidence intervals.

4.3 Quality of the final configurations

In the network of Figure 3, there exist two stable configurations (in other words Nash equilibria) where no flow can lower its delay by changing its path (these two configurations are given in Figure 4). By Theorem 1, the OPS Algorithm converges to either of these configurations.

Only Flows 2 and 4 differ between these two configurations. In the second configuration, both flows choose a longer path. Thus, the first configuration is better than the second one, because the average delay of any flow is either smaller or unchanged. However, both configurations successfully avoid congestion so that they both constitute good choices (even if not optimal).

The following table shows the percentage of experiments that converged to equilibrium 1, equilibrium 2, any non-stable configuration, or did not converge. As the load increases, it can be observed that equilibrium 1 (the best configuration) is more likely to be selected by our algorithm, which matches intuition.

Load	0.7	0.8	0.875	0.9	0.925	0.95	0.975
Eq. 1	54.1	56.5	65.9	68.2	76.5	83.5	58.3
Eq. 2	44.7	43.5	31.8	31.8	21.2	15.3	17.9
Other	1.2	0.0	2.4	0.0	2.4	1.2	13.1
No cv.	0.0	0.0	0.0	0.0	0.0	0.0	10.7

For each value of network load, 85 independent experiments were conducted. The results in the table use our convergence criterion with $\Theta = 0.8$ and $W = 50$. The executions reported as non-convergent had failed to converge after the end of the experiment (2400 seconds).

The quality of the convergence actually depends on the parameters Θ and W. If one chooses the threshold Θ close to one, then the quality improves, in the sense that the number of times that the OPS algorithm stops under an unstable configuration drops. However this comes at a cost: the time of convergence increases as well as the number of executions that did not converge. An empirical search (not reported here) with varying values of Θ and W shows that a good compromise in our experiments is obtained by choosing $\Theta = 0.8$ and $W = 50$.

5 Conclusion

We have described an adaptive routing algorithm for packet-switched networks, first as a theoretical algorithm, and then as a practical implementation that can be used as a Openflow controller. We have first proved a convergence theorem in a idealized network model, showing that our algorithm converges to a quasi-Nash Equilibrium. Using our prototype implementation in an emulated network, we have then shown that our SDN implementation converges fast under light to moderate network load, and indeed reaches a Nash Equilibrium (i.e. a stable configuration where no flow can improve its delay). Under very heavy network load, convergence is slower, which is not surprising given the variability of delays when the network becomes completely saturated. However, even in these extreme conditions, our implementation still converges to a quasi-Nash Equilibrium in most cases.

6 Future work

6.1 Dynamic network environment

Our current implementation uses static routing tables. An obvious improvement would be to couple our controller with a routing protocol, such as BGP or OSPF, that would compute possible paths for each destination and feed them to our algorithm.

This brings up the question of dynamic network conditions, which would create new paths or invalidate existing paths in the network. In addition, flows can have varying demand over time. For now, we have focused on routing over a stationary network. If the network becomes dynamic, the proof of convergence would be more difficult, and can no longer be based on the theoretical tools we used. One promising direction is to use two time-scale stochastic approximations: On the slow time scale, network conditions and flow demand can evolve. On the fast time-scale, the algorithm updates its scores and path choices for each packet.

6.2 Congestion control

Let us consider a more challenging situation, where flow endpoints use a congestion control algorithm. This means that the throughput may depend on the routing choices, since the packet loss, available capacity and delay for a flow all depend on the path used by the flow.

This raises the question of the interaction between congestion control at endpoints and routing decision made by our algorithm. This can possibly lead to a "self-confirming" situation as described by [4], where congestion control reduces throughput because of bottlenecks created by poor routing choices, leading to this stable but inefficient situation. In [4] the authors further prove that an adaptive routing algorithm that minimizes the maximum link utilization (the so-called "min-max" property) converges to a stable and optimal situation, given some fairness assumptions on the congestion control mechanism.

Thus, we leave to future work the stability and optimality considerations related to congestion control; for instance, one could study situations and hypotheses for which our algorithm satisfies the min-max property.

A Appendix: Proof of Theorem 1

This appendix is devoted to the proof of Theorem 1. The OPS algorithm is a variant of an algorithm presented in [15], that computes Nash equilibria in potential games. First, our routing problem can be seen as a game. The flows are the players, their strategies are the choices of paths and the payoffs (or the costs, here) are the expected delays on the chosen paths. Under this framework, our problem fits in a well-known class of games, namely *atomic routing games*. In [25], it was shown that atomic routing games where each player (flow) has rate 1 are *potential games*, *i.e.* there exists a *potential function* for such games, namely $\phi(\mathbf{p}) = \sum_e \sum_{i=1}^{N_e(\mathbf{p})} \delta_e(i)$. Here, the rates are not necessarily equal to one but they can easily be replaced by flows with rate one by changing units. This is where Assumption 2 is used in our case. It implies that our problem is a potential game.

Using this construction, our algorithm is similar to Algorithm 1b in [15] with the addition of bounding terms β_n.

The L2-L1 condition on γ_n is the same as assumption A1 in [15]. Our Assumption 1 implies Assumption A2 in [15]. As for assumption A3 in [15], it is replaced here by the explicit bounds β_n. The bounded approximation theorem (Theorem 2) in [5] allows us to state that the sequence Y_n is an asymptotic pseudo-trajectory (APT) in the sense of [9] as soon as the clock ticks for the flows all have finite rates. This implies that for each flow the scores of paths, Y_p, approach the solution of the following differential system (5)-(6), even if the estimation D is based on outdated measurements of packet delays

$$\frac{dy_p}{dt} = d_p(\mathbf{q}) - \tau y_p, \quad \forall p \in \mathcal{P} \tag{5}$$

$$q_p = \frac{\exp(y_p)}{\sum_\ell \exp(y_\ell)} \quad \forall p \in \mathcal{P}, \tag{6}$$

as long as this solution is locally stable (in the dynamical system sense).

In [15], is is shown that the solutions of this differential system are locally stable when the game is a potential game and are $\epsilon(\tau)$ approximations of Nash equilibria of the game, where the error $\epsilon(\tau)$ vanishes as $\tau \to 0$. This means that Theorem 2 in [5] can be used here.

Finally, the existence of a potential function further implies that the only locally stable Nash equilibria are pure (see [23, 28]). In our case this means that there exist a path p_k for each flow k such that

$$d_k(p_1, \ldots, p_k, \ldots, p_K) \leqslant d_k(p_1, \ldots, p', \ldots, p_K),$$

for all p' in $\mathcal{P}_k$. Therefore our algorithm will converge arbitrarily close to such a pure Nash equilibrium. This concludes the proof.

References

[1] Ryu: a component-based software defined networking framework. http://osrg.github.io/ryu/.

[2] J. Abley, B. Black, and V. Gill. Goals for IPv6 Site-Multihoming Architectures. RFC 3582 (Informational), August 2003.

[3] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review*, 38(4):63–74, 2008.

[4] Eric J Anderson and Thomas E Anderson. On the stability of adaptive routing in the presence of congestion control. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 2, pages 948–958. IEEE, 2003.

[5] Sigrún Andradóttir. A stochastic approximation algorithm with varying bounds. *Operations Research*, 43(6):1037–1048, 1995.

[6] Baruch Awerbuch and Robert D Kleinberg. Adaptive routing with end-to-end feedback: Distributed learning and geometric approaches. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 45–53. ACM, 2004.

[7] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[8] Sebastien Barre, Christoph Paasch, and Olivier Bonaventure. MultiPath TCP: From theory to practice. In Jordi Domingo-Pascual, Pietro Manzoni, Sergio Palazzo, Ana Pont, and Caterina Scoglio, editors, *NETWORKING 2011*, volume 6640 of *Lecture Notes in Computer Science*, pages 444–457. Springer Berlin Heidelberg, 2011.

[9] Michel Benaïm. Dynamics of stochastic approximations. In *Séminaire de Probabilités*, volume 1709 of *Lectures Notes in Mathematics*, pages 1–68, 1999.

[10] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Machine Learning*, 5(1):1–122, 2012.

[11] Chiara Chirichella, Davide Rossi, Claudio Testa, Timur Friedman, and Antonio Pescape. Passive bufferbloat measurement exploiting transport layer information. In *Global Communications Conference (GLOBECOM), 2013 IEEE*, pages 2963–2968. IEEE, 2013.

[12] George Christodoulou and Elias Koutsoupias. The price of anarchy of finite congestion games. In *37th Annual ACM Symposium on Theory of Computing (STOC)*, 2005.

[13] R. Cole and Y. Tao. The price of anarchy of large walrasian auctions. Technical Report arXiv:1508.07370v, ArXiv, 2015.

[14] Riccardo Colini-Baldeschi, Roberto Cominetti, and Marco Scarsini. On the price of anarchy of highly congested nonatomic network games. In Springer, editor, *9th International Symposium of Algorithmic Game Theory (SAGT)*, number 9928 in LNCS, pages 117–128, 2016.

[15] Pierre Coucheney, Bruno Gaujal, and Panayotis Mertikopoulos. Penalty-Regulated Dynamics and Robust Learning Procedures in Games. *Mathematics of Operations Research*, 40(3):611–633, 2015.

[16] Wenzhi Cui and Chen Qian. Difs: Distributed flow scheduling for adaptive routing in hierarchical data center networks. In *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, pages 53–64. ACM, 2014.

[17] Bernard Fortz and Mikkel Thorup. Optimizing ospf/is–is weights in a changing world. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 20(4), 2002.

[18] T. C. Hu. Multi-commodity network flows. *Operations Research*, 11(3):344–360, 1963.

[19] Atul Khanna and John Zinky. The revised arpanet routing metric. *ACM SIGCOMM Computer Communication Review*, 19(4):45–56, 1989.

[20] Walid Krichene, Benjamin Drighès, and Alexandre Bayen. On the convergence of no-regret learning in selfish routing. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 163–171, 2014.

[21] Amund Kvalbein, Constantine Dovrolis, and Chidambaram Muthu. Multipath load-adaptive routing: Putting the emphasis on robustness and simplicity. In *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 203–212. IEEE, 2009.

[22] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[23] Dov Monderer and Lloyd Shapley. Potential games. *Games and economic behavior, Elsevier*, 14(1):124–143, 1996.

[24] Arkadi Semen Nemirovski and David Berkovich Yudin. *Problem Complexity and Method Efficiency in Optimization*. Wiley, New York, NY, 1983.

[25] A. Orda, R. Rom, and N. Shimkin. Competitive routing in multiuser communication networks. *IEEE/ACM Trans. on Networking*, 1(5):510–521, 1993.

[26] T. Roughgarden and E. Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2):239–259, 2002.

[27] Gilles Berger Sabbatel and Martin Heusse. ipmt: Internet protocols measurement tools. https://forge.imag.fr/projects/ipmt/.

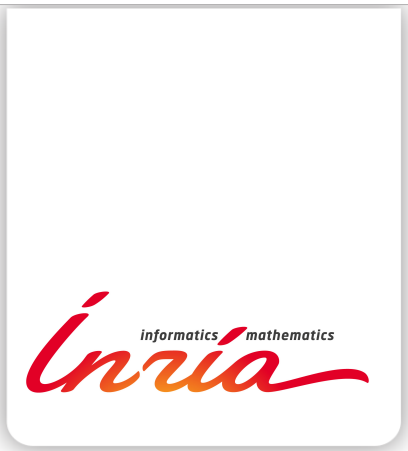
[28] William H. Sandholm. *Population Games and Evolutionary Dynamics*. MIT Press, 2010.

[29] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817 (Experimental), December 2012.

[30] O. Troan, D. Miles, S. Matsushima, T. Okimoto, and D. Wing. IPv6 Multihoming without Network Address Translation. RFC 7157 (Informational), March 2014.

[31] Bryan Veal, Kang Li, and David Lowenthal. New methods for passive estimation of tcp round-trip times. In *Passive and Active Network Measurement*, pages 121–134. Springer, 2005.

[32] Xin Wu and Xiaowei Yang. Dard: Distributed adaptive routing for datacenter networks. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 32–41. IEEE, 2012.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399